EXPRESSIVE AND SCALABLE EVENT STREAM PROCESSING

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Mingsheng Hong

January 2009

EXPRESSIVE AND SCALABLE

EVENT STREAM PROCESSING

Mingsheng Hong, Ph.D.

Cornell University 2009

Rapid technical advances have made it possible to instrument even massive computing systems. However, the technology for processing high-speed data streams from physical sensors and software systems has lagged the capability to produce such streams. The goal of stream processing research is to develop algorithms and software infrastructures capable of processing streaming data with high throughput and low latency.

A large class of streaming applications requires that stream processing systems be both expressive and scalable. That is, a stream processing system should be able to process a large number of reasonably sophisticated queries over high-speed input streams. There are however general tensions between expressiveness and scalability in stream processing. In this dissertation, we present techniques and prototype systems that address both expressiveness and scalability aspects.

First, we present Cayuga, a general-purpose event monitoring system with an expressive event algebra and a set of novel query optimization techniques. Second, we describe a rule-based Multi-Query Optimization framework, which generalizes Cayuga, unifies large-scale stream processing and event processing, and provides a platform for integrating future query rewrite based optimization techniques. Finally, we describes an approach in large-scale XML stream join processing. To our knowledge, this is the first scalable solution to the problem of processing a large number of XML stream join queries.

As the adoption of stream processing technology increasingly gains momentum, the

ideas and techniques developed in this dissertation provide a foundation for building expressive and scalable stream processing systems with affordable cost and high reliability.

BIOGRAPHICAL SKETCH

Mingsheng Hong is a Ph.D. student in the Department of Computer Science at Cornell University. Mingsheng's research interests are in the areas of data stream and complex event processing. Together with colleagues at Cornell, Mingsheng has built Cayuga, an expressive and scalable event monitoring engine. Mingsheng is also one of the founders of the CEDR event processing project at Microsoft Research, Redmond. In the past three summers at Redmond, he co-developed an expressive temporal stream computation model, a declarative event pattern query language, and a set of run-time operators with efficiency guarantee.

ACKNOWLEDGEMENTS

I would like to first thank my advisor Professor Johannes Gehrke for his guidance and support during my graduate study at Cornell University, without which I could not have completed this dissertation. He brought me into the world of database research, and taught me many skills essential for high-quality research work, including how to read, think, and write critically, how to give clear presentations, and how to conduct effective experimental studies. I am grateful for the numerous meetings that he had with me, in which he provided invaluable guidance in research topics, techniques and agenda. In addition to the research skills, I am also greatly influenced by his style and manners in communicating with students and research colleagues, and his serious and high-standard attitude towards work.

I would like to express my sincere gratitude to my colleagues in the Cornell Cayuga team. I especially want to thank Dr. Alan Demers for spending countless number of hours with me discussing system design and implementation issues, enlightening me with his insights in building efficient and robust systems. Special thanks also to Dr. Mirek Riedewald for his devoted efforts in providing guidance and feedback to my thesis work. I also thank Dr. Walker White for the discussions on the Cayuga project. I am proud to be part of the Cayuga team working with them.

I also want to acknowledge Professor Richard Shore for providing me with guidance on Mathematics study, for teaching me the interesting and challenging subject of Recursion Theory, and for participating in my thesis committee.

I am also grateful to my three mentors in the Microsoft Database Research Group, David Lomet, Roger Barga, and Jonathan Goldstein, for their teachings, encouragement and support during my summer internships at MSR.

Last but not least, I am deeply indebted to my wife Yankun and my parents for their unconditional love and encouraging support. They gave me the strength to overcome the

obstacles that I encountered during my PhD study. I dedicate this dissertation to them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Rapid technical advances have made it possible to instrument even massive computing systems. However, the technology for processing high-speed data streams from physical sensors and software systems has lagged the capability to produce such streams. The pressing need is for software infrastructures capable of processing streaming data with high throughput and low latency. Such software infrastructures are usually referred to as *stream processing systems*.

Advances in stream processing technology have been driven by a large class of both well-established and emerging applications, including supply chain management for RFID (Radio Frequency Identification) tagged products, real-time stock trading, monitoring of large computing systems to detect malfunctioning or attacks, and monitoring of sensor networks, e.g., for surveillance. There is great interest in these applications as indicated by the establishment of sites like `http://www.complexevents.com`, which bring together major industrial players like BEA, IBM, Oracle, and TIBCO. A common characteristics among these applications is the need to answer reasonably sophiscated queries over massive streams in (near) real-time.

As a result of such high demands from the industry on stream processing technology, start-up companies such as StreamBase [86] and Coral8 [28] have had initial success in offering commercial products of stream processing systems. In addition, traditional software leaders such as IBM and Microsoft are also beginning to roll out their solutions in stream processing [50, 64].

1.1 Focus and Main Contributions of Dissertation

At the initial stage of this thesis work, there has been various research projects focusing on the foundational issues on stream processing, including the development of stream processing model, the semantics of common stream processing operators, and the system architecture design of a high speed streaming system [77, 24, 67, 21]. In addition, research work on publish/subscribe systems conducted during the 1990s is also very relevant to stream processing.

Among the existing literature in this rich area of research, we observed the following *expressiveness-scalability dichotomy* between theoretical and systems-oriented approaches. Theoretical approaches, based on formal languages and well-defined semantics, result in expressive query algebras and languages, but generally lack efficient, scalable implementations. On the other hand, the scalability proposals in systems approaches usually focus on query languages of very limited expressiveness. As a result of this observation, it has become the main theme of this thesis research to address the stream processing problem from the angles of both expressiveness *and* scalability simultaneously.

In developing Cayuga, we have taken great care to define a language that can express very powerful subscriptions, has a precise formal semantics, and can be implemented efficiently. The resulting Cayuga algebra is very different from Aurora's boxes-and-arrows approach and SQL-based languages like STREAM's CQL [11]. Following the development of the Cayuga algebra, we addressed the challenges in the design and implementation of the Cayuga system, and demonstrated the efficacy of our approach through extensive experimental study.

Our work on Cayuga pointed us to another interesting direction of research: to ex-

plore the commonalities between event processing and stream processing, and determine how to combine the strengths of each of them. This result of this investigation is a rule-based Multi-Query Optimization framework, which generalizes Cayuga, unifies large-scale stream processing and event processing, and provides a platform for integrating future query rewrite based optimization techniques.

Our observation on the expressiveness-scalability dichotomy applies not only to relational streams, but to XML streams as well. Although there has been work on efficient and scalable XPath stream processing involving few or no joins, as well as expressive XQuery stream processing involving few queries, we are the first to develop a large-scale processing scheme for evaluating a large number of join queries over XML streams.

1.2 Outline of this Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 provides the technical background of this dissertation, and covers related work. Chapters 3-6 contain the contributions of this thesis. We split the presentation of Cayuga into two chapters. Chapter 3 describes the formal foundation of Cayuga, including the event model, algebra, and processing model. The design and implementation challenges of the Cayuga system, as well as the novel query optimization techniques in the Cayuga query engine, are discussed in Chapter 4. Chapter 5 presents a rule-based Multi-Query Optimization framework generalizing Cayuga. Chapter 6 describes our approach in large-scale XML stream join processing. Chapter 7 concludes this dissertation.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 Introduction To Data Stream Processing

Since data stream processing is a relatively young field, there is no established data model like the relational model for relational databases. However, there is consensus on some aspects of an appropriate stream model as follows. First, a stream is an infinite set of tuples or *events*[1], where there exists a temporal ordering (partial or total order) among these tuples, denoted as $\prec$. Thus, in addition to the usual data fields, stream tuples also carry time stamps, which establish such a temporal ordering. Second, any sensible operation on a stream must process the stream incrementally w.r.t. the temporal ordering. That is, the operation cannot expect to see the entire stream before it produces any result. Third, any sensible operation on a stream can only read the stream in one pass. This is motivated by practical concerns that streams are usually generated on real-time. Note that the operation can still have its internal memory to remember a *summary* of the stream tuples it has read, but it cannot remember *all* the stream tuples it has read, since that would require an infinite amount of storage. Therefore, assuming a stream consists of a sequence of integers, then operations such as sorting that entire stream, or computing median on that entire stream, are excluded from stream processing.

Formally, we can define a stream S as an infinite set $\{(d_1; t_1), (d_2; t_2), \ldots\}$, where the d_i and t_i are the payload component (data fields) and the timestamp component of a stream tuple, respectively. It is generally agreed that the payload component can be modeled by a relational schema fixed for each stream. However, there are different designs for the timestamp component, which results in different semantics and expressive

[1] In the following text, we use the term tuple and event interchangeably

power of the stream models. For example, some stream models can allow for simultaneous tuples (tuples that occur at the same time), while others cannot. Also, in some stream models $\prec$ is a total order, while in others it is a partial order. We will focus on existing approaches to modeling time in Section 2.2.

The design of stream query languages has to be constrained by the particular properties of streams described above. In general, people would like to design a query language such that the queries expressed in it have the following desirable properties. P1: The queries should produce high quality of output (e.g. exact answers will be of higher quality than approximate answers). P2: User should be able to formulate powerful queries (the query language should be expressive). P3: The queries should be efficient to evaluate in terms of space and time complexity. Clearly there are operations that do not satisfy all of the three properties, such as computing median on the stream of integers, since it violates P3. To ensure P3 holds, either P1 or P2 will usually have to be sacrificed. By sacrificing P1, researchers could restrict the input scope of query operations, so that the stream queries can be processed reasonably efficiently. This is usually done through various forms of *window* operators, which read streams as inputs, and produce a finite set of stream tuples for other query operations to consume. For example, instead of computing median on an entire stream of integers, we could instead compute the median value of a window of the 10 last integers we saw in the stream. In Section 2.3 we present the representatives of stream query languages that make such a tradeoff, and the corresponding query processing techniques will be described in Section 2.4.

By sacrificing P2, researchers have developed approximate algorithms for solving various problems on stream processing. These algorithms are outside the scope of this work.

2.2 How to Model Time

Recall from Section 2.1 that a stream is an infinite set of tuples, where each tuple carries an explicit timestamp. Here we describe a set of representative timestamp models.

2.2.1 Point Timestamp Model

In the timestamp model adopted by the Stanford STREAM system [14], each tuple has a point timestamp. Thus the domain of timestamps is isomorphic to the set of natural numbers. The temporal ordering $\prec$ on the domain of point timestamps is therefore a total order. This timestamp model is suitable for many streaming scenarios. For example, each reading generated by a sensor can be naturally modeled as a stream tuple under this timestamp model, where the timestamp value of that tuple is assigned by the sensor which generates the reading. This timestamp model is adopted by other stream systems, such as Aurora, Borealis, TelegraphCQ and SASE [21, 3, 24, 95]. Among them, some systems such as STREAM allow streams to have *simultaneous* tuples; that is, multiple tuples with the same timestamp value. Others systems, such as SASE, do not allow simultaneous tuples.

This timestamp model has the advantage of being simple to use. When specifying the semantics of an operator on a set of input streams, it is usually fairly straightforward to express what kind of temporal constraint the operator imposes on its input stream tuples, and what the timestamp values for the output tuples produced by the operator should be. For example, a selection operator reads one input stream, and outputs a subset of tuples from the input stream that pass the selection condition. In its semantics, it imposes no temporal constraint on the input tuples, and the timestamp of each output

tuple is set to be the same as its corresponding input tuple that passes this selection. On the other hand, a binary sequence operator reading two input streams S_1 and S_2, which we denote as $S_1; S_2$ using infix notation, produces an output event when there is an S_1 tuple denoted as e_1, and an S_2 tuple denoted as e_2, such that the timestamp value of e_2 is greater than that of e_1 in terms of $\prec$ (we say e_2 *follows* e_1 in this case). This is the temporal constraint the sequence operator imposes on its input stream tuples. For that output event, its timestamp value is set to that of the e_2.

However, for some more complex streaming scenarios, where stream tuples may have nontrivial time durations, assigning a point timestamp to each stream tuple may become inadequate. As an example, consider a stream tuple representing a coupon that is valid for a certain period of time. The point timestamp model will not be able to represent the validity interval of this coupon with a single stream tuple – either such an interval has to be encoded by two stream tuples under this model, or it will have to be encoded in the payload component. In either case, it makes it hard to formulate certain queries, such as the query that counts the number of valid coupons at a certain time point or continuously over all time points. Another limitation of the point timestamp model is that the semantics of certain operators defined under this model could become dubious. For example, for the binary sequence operator, it can be verified that its semantics under the point timestamp model we introduced above demonstrates the following unintuitive semantics: $S_1; (S_2; S_3) \equiv S_2; (S_1; S_3)$[2]. This is unintuitive because we expect each output event from expression $S_1; (S_2; S_3)$ to consist of an S_1 event followed by an S_2 event, which is then followed by an S_3 event. The above equivalence rule clearly states otherwise, that the ordering of the S_1 event and the S_2 event that form the output event is immaterial.

[2]Two streams $S \equiv S'$ iff they have exactly the same set of tuples.

2.2.2 Interval Timestamp Model

More sophisticated timestamp models that use interval timestamps address the above two limitations of the point timestamp model. Cayuga, which we will present in the next two chapters, uses such a model. Here we provide an overview. In the Cayuga timestamp model, each stream tuple takes the form of $< \bar{a}; t_0, t_1 >$, where $\bar{a}$ denotes the payload component with a relational schema, and t_0 and t_1 respectively denote the start and end time of the interval timestamp of this tuple. Tuples are assumed to occur in the stream at their t_1 time.

An interval timestamp model allows us to naturally encode real-world objects with nontrivial temporal durations as stream tuples. Also, the operators defined under this model could have more intuitive and richer semantics. However, under the interval timestamp model, it becomes nontrivial to define the semantics of some operators appropriately. This is partly because the temporal ordering $\prec$ on the domain of interval timestamps becomes a partial order. To illustrate some of the difficulties in giving an appropriate semantics of an operator, let us revisit the binary sequence operator introduced in Section 2.2.1. We would like to define the sequence operator under the interval timestamp model in such a way that the following requirements are met. First, the interval timestamp of an output event by the sequence operator should minimally cover the interval timestamps of its input events that form this output event. Second, the sequence operator should not exhibit unintuitive semantics as in $S_1; (S_2; S_3) \equiv S_2; (S_1; S_3)$ under the point timestamp model. Third, for each S_1 event e_1, the sequence operator as in $S_1; S_2$ should only pick the *first* S_2 event that qualifies for producing an output event together with e_1. This is to reduce the output volume of a sequence operator so that the consumer of its output stream is not overwhelmed. For example, consider the scenario where there is an S_1 event followed by 100 S_2 events. It is likely that the user only

intends to catch *one* instance of such sequencing in his/her use of the sequence operator, instead of *all* instances, which could be combinatorially large.

The definition of sequence operator in Cayuga satisfies the above statements. Here we informally illustrate its semantics with an example. Its formal semantics will be described in the next chapter. Let S_1 and S_2 be the two input streams of the sequence operator, as shown in Figure 2.1. Its output stream consists of one tuple, formed by the combination of e_1 and e_4. the interval timestamp of that output tuple starts at the start time of e_1, and ends at the end time of e_4. The sequence operator definition in Cayuga has the following additional properties. It deals with simultaneous events in a deterministic way. Also, it allows an optional predicate parameter θ, such that for an S_1 event e_1, to produce an output event for $S_1;_\theta S_2$, it can *skip* those S_2 events that temporally follow e_1 but do not satisfy θ together with e_1. This useful feature allows the sequence operator to support queries like the following: for the current stock quote in IBM, find me the price of the next IBM quote. If the input stock quote stream contains quotes for other companies between the current IBM quote and the next, the sequence operator will be able to skip these quotes and only match the next IBM quote with the current one.

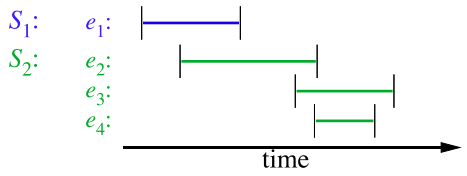


Figure 2.1: Time intervals of arriving events

In the point time model and interval time model we introduced so far, we have been focusing on only one notion of time – the time from event sources' perspective. We refer to this notion of time as *application time*. We refer to a timestamp model which

only has the notion of application time as a *unitemporal model*. Unitemporal models are widely adopted by existing stream processing projects, and they serve well in defining the query operator semantics. However, unitemporal models are not flexible in dealing with out-of-order event delivery.

We describe out-of-order delivery as follows. The temporal ordering $\prec$ introduced above is defined on application time. Semantically, any operation on streams is to read stream tuples in the order given by $\prec$, and produce incremental results. If the stream processing system were to stay in the same machine host as the event sources, the processing system could read the tuples generated by the sources in exactly the same order, and produce output according to the query operator semantics. However, in reality, the stream processing system may live in a separate machine host from the event sources. The underlying network between the processing system and the event sources may delay the delivery of events to the processing system, such that some events may appear out-of-order when they arrive at the processing system.

In Section 2.2.3 we will describe some techniques in dealing with out-of-order events under the unitemporal model. Section 2.2.4 describes a timestamp model that allows for more flexible techniques to address the same issue. This degree of flexibility is achieved by exploring temporal dimensions beyond the application time.

2.2.3 Dealing with Out-Of-Order Events in Unitemporal Model

In a unitemporal model, a common strategy that the processing system can use to handle out-of-order event delivery while guaranteeing the correctness of query outputs is to buffer the input events and attempt to re-establish the temporal ordering $\prec$ on application time, before processing these input events. Knowledge external to the processing system

is required in this event reordering process. Such knowledge usually takes the form of special events in the stream, referred to as *heartbeats* or *punctuations* in the stream processing literature [84, 89]. We informally describe the the semantics of these special events as follows[3]. When the processing system receives a heartbeat event of value t on a stream S, it is guaranteed to have received all events whose timestamps are no more than t. Here we describe how the processing system can use heartbeats to reorder events and produce correct query outputs. We assume the processing system reads only one input stream. This technique can however be generalized to deal with any finite number of input streams. When the processing system reads input events, it buffers all of them. Whenever it sees a heartbeat of value t, it could process its buffered events in the order given by $\prec$ up to timestamp value t. This way the processing system is guaranteed to process events in the order as they are generated by event sources.

Heartbeats can be generated directly from event sources. In this case, the underlying network between the event sources and the processing system should guarantee that the semantics of heartbeats are preserved during data transmission. For example, heartbeats cannot be reordered with other regular events in the data transmission. Alternatively, heartbeats can be generated by a third party. For example, [84] proposes techniques to automatically *deduce* heartbeats from the knowledge of the physical stream processing environment, such as the maximum clock skew between the event sources and the processing system, and the upper bound for network delay. In either case, the processing of events will have to be blocked from time to time when the system is expecting incoming heartbeats. As a result, the throughput of the processing system may be negatively impacted, since the system may be in an idle state, even if there are events waiting to be processed. Such a waste of computing resources is undesirable. Clearly, the processing system can block less frequently if the frequency of heartbeats on the stream can be in-

[3]We restrict the discussion of heartbeats to point timestamp models. Their semantics under interval timestamp models is similar.

creased. However, putting too many heartbeats on the stream also wastes the computing resources of network bandwidth and the processing time. Therefore for a given workload, the frequency of heartbeats of the stream may have to be tuned to achieve maximal system throughput. It is an open problem as to how to effectively tune such a parameter.

2.2.4 Beyond Application Time

The fundamental limitation of unitemporal models in dealing with out-of-order delivery lies in the fact that incoming events cannot be processed by the system before the temporal ordering is re-established. The Bitemporal stream model proposed by the Microsoft CEDR project [18] addresses this problem by allowing event processing to proceed before the temporal ordering on application time is re-established on the processing system's side. As such, it is possible for the processing system to issue incorrect output sometimes. However, the bitemporal stream model allows the system to produce "corrections" to previous outputs, such that the eventual state of the output stream produced by the system matches the query semantics. Such a query processing strategy is referred to as *optimistic query processing*, and it effectively unblocks the query processing system. In comparison, correction events cannot be encoded under a unitemporal model, and therefore optimistic query processing cannot be directly applied in that setting.

So far we have described two processing strategies for handling out-of-order events. They form two extremes on the spectrum of processing strategies. CEDR identifies and formalizes such a spectrum, which it refers to as *consistency guarantees*. In this section, we informally describe this bitemporal stream model of CEDR, as well as the set of consistency guarantees. We refer the reader to [18] for the technical formalism and details.

In the bitemporal stream model, besides application time, we also model *system time*, which is the notion of time from the stream processing system's perspective. Each stream tuple now have two timestamps, an interval timestamp on the application time, denoted as *validity interval*, and a point timestamp on the system time, denoted as *CEDR time*. The latter explicit timestamp on system time allows us to model out-of-order arrival of stream tuples.

Based on the bitemporal stream model, CEDR formally defines a set of consistency guarantees to be associated with queries. Here we illustrate these guarantees through examples of monitoring queries in financial markets. In scenario 1, we have queries running in a compliance office to monitor trader activity and customer accounts, in order to ensure conformity with SEC rules and institution guidelines. To make accurate assessment, these queries have strict requirements on the correctness and stability of output. That is, the output should conform to the query semantics, and should not contain any corrections. As a result, events will have to be processed in the same order as they are generated by event sources. We refer to this processing strategy as *strong consistency*. In scenario 2, we have queries running in trading floors to extract events from news feeds and correlate with market indicators, possibly impacting the automated stock trading programs. The requirements for these queries are that they should be highly responsive, and the output should conform to the query semantics, but they can be allowed to have corrections, since trades are sometimes allowed to be retracted in financial markets. As a result, the processing system should process incoming events as early as possible, but retain enough state information that later when it detects some early mistakes in the output, it can issue corrections. We refer to this processing strategy as *middle consistency*. Finally, in scenario 3, we have queries running on a trader's desktop to track a moving average of the value of an investment portfolio. Such queries need to be highly responsive, but do not require perfect accuracy of the output. Consequently,

the processing system can process incoming events as early as possible, and does not have to keep enough states to correct its output. We refer to this processing strategy as *weak consistency*.

There are various trade-offs associated with the different consistency guarantees above. The first set of trade-offs is between the degree of blocking in the processing system and the amount of memory resource the processing system needs, as shown in Figure 2.2. On the extreme corresponding to strong consistency, the system needs a lot of memory to buffer incoming events so that it can re-establish the temporal ordering given by the event sources. Since events cannot be processed before the system is sure of the correct ordering among them, the system will have to be blocked fairly often. On the other extreme corresponding to weak consistency, since the system processes incoming events as soon as possible, it does not block, and since the system is not required to correct its past output if there are errors, it does not need much memory to maintain the query states. Finally, we have yet another extreme corresponding to middle consistency, where the system is allowed to produce output as soon as incoming events arrive, and therefore it does not block; on the other hand, since it needs to correct its past output when out-of-order events causes incorrect query output, it needs a lot of memory to maintain the query states. Note that any "mixed" consistency degree corresponding to a point in the shaded triangle region in Figure 2.2 is possible.

Another set of trade-offs is between high quality of output, non-blocking of the system, and small size of the output streams produced by the system. These three desirable properties are denoted by the three end points of the triangle shown in Figure 2.3. Different degrees of consistency are drawn on the triangle edges. Each consistency degree can only achieve two out of the three nice properties. For each consistency degree in Figure 2.3, its two achievable properties correspond to the two triangle nodes connected by the

edge corresponding to that consistency degree. For example, for strong consistency, the output streams have high quality and minimum size, since there are no correction events at all. however, the system will have to block from time to time.

Table 2.1 puts together these different trade-offs.

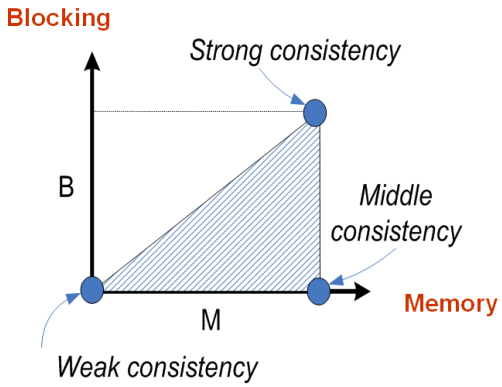


Figure 2.2: Trade-off 1

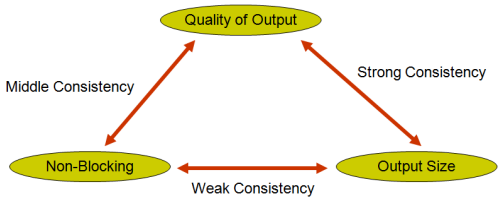


Figure 2.3: Trade-off 2

Table 2.1: Consistency Trade-Offs

Consistency	**Quality of Output**	**Blocking**	**State Size**	**Output Size**
Strong	High	Yes	High	Low
Middle	Middle	No	High	High
Weak	Low	No	Low	Low

Finally, we make a concluding remark on the bitemporal stream model with some connections with techniques in database transaction management. Each transaction can be associated with an isolation level, which trades off the "consistency" of this transaction with the degree of concurrency the transaction processing system is able to achieve. For example, a transaction running in the isolation level of READ UNCOMMITTED gives maximal concurrency, but the database state after running this transaction could be

inconsistent. Similarly, under the bitemporal stream model, each query could be associated with a consistency degree, which exhibits various trade-offs shown in Figure 2.1. Also, in the bitemporal stream model and CEDR, the technique of optimistic query processing may increase the throughput of the stream processing system by unblocking the processing of incoming events, but this may also cause the system to issue occasional corrections. Similarly, in transaction processing, the technique of optimistic concurrency control [56] increases concurrency of the transaction processing system, but some transactions may have to be rolled back if conflicts of read and write operations among different concurrent transactions are detected.

2.3 Stream Query Language

A stream query language is the interface between the stream processing system and its clients. Such clients could be application programs or end users. In this section, we define a query language in a broad sense. A query language could take various forms. First, it could be specified in user-friendly syntax with key words drawn from natural languages, such as SQL for relational query processing. The CQL language developed by STREAM project, which we will describe in Section 2.3.1, falls into this category. Second, a query language could consist of a set of operator boxes, which users can choose and connect with edges to form query plans. Such a query language is much like a general Graphic User Interface for end users to communicate with the computer software systems. The Aurora and Borealis query language, to be described in Section 2.3.2, takes this form. Finally, a query language could be an algebra consisting of a set of operators represented by mathematical symbols, such as the relational algebra for relational query processing. The Cayuga stream algebra, to be presented in the next chapter, is one such example.

2.3.1 STREAM's Continuous Query Language

SQL and its underlying relational algebra [72] have proven very successful for querying datasets. Their semantics are well-understood and users are comfortable formulating queries in SQL. Hence it is natural to attempt to leverage as much of SQL and relational algebra as possible for querying data streams.

As Law et al. [57] show, SQL lacks expressive power for continuous queries on data streams. Wang et al. address this deficiency by extending SQL with features to support data mining and data streams [91]. Other work that leverages the relational model includes [24, 25, 44, 11].

In the following we present an overview of CQL, the query language of Stanford's STREAM system [67].

Like SQL itself, CQL is declarative and admits of a formal specification [11]; and there are some initial results characterizing a sub-class of queries that can be computed with bounded memory [85, 10]. CQL extends SQL with operators that read or write streams. These operators work as adapters to convert streams into relations, and vice versa. Continuous queries on data streams read input streams, and produce new streams. However, CQL does not have native stream-to-stream operators. Therefore, each query must consist of some stream-to-relation operator(s) to convert input streams into relations first, followed by a query expression using only SQL operators, followed by some relation-to-stream operator to produce an output stream.

The stream-to-relation operators in CQL are referred to as window operators. A time-based window operator is associated with a parameter T. It continuously "slides" over the input stream, and at (application) time t, it produces a relation consisting of tuples with timestamps within the range between $t - T$ and t. Similarly, a tuple-based

window operator is associated with a parameter N. At time t, it produces a relation consisting of the most recent N tuples in the stream. CQL has three relation-to-stream operators, which all convert their input relations at each time instance t into a set of stream tuples with timestamp t. Details can be found in [11].

Since CQL is based on SQL, a relation in CQL is an (unordered) set of tuples. As a result, during query processing, the temporal ordering of tuples obtained from the input streams may be lost, and it is expensive to recover such ordering information. Specifically, consider the relation produced by a window operator over the last N tuples in the input stream. In order to pinpoint the i-th tuple (in terms of temporal ordering) in this relation, where i is a number between 1 and N, an N-way self-join with temporal constraints on these N tuples is required. This makes it unnatural to express certain stream queries, and the processing cost of those queries may be unnecessarily increased.

2.3.2 Aurora/Borealis Query Plans

The Aurora/Borealis system [21, 3] takes a very different approach compared to CQL. The authors argue that querying streams, which usually means that streams are *monitored*, is significantly different from the relational model [21]. Hence they propose to design the query language from ground up.

Aurora/Borealis use a procedural boxes-and-arrows paradigm. Queries can be composed of arbitrary data processing "black boxes", which are connected by data pipelines through which the tuples flow from one operator to the next. A variety of pre-defined operators are available. When formulating a query, the user will drag and drop a set of operator boxes from a "toolbox" window in a graphic development environment, and connect them to form a data flow.

Such a box-and-arrow style language is at the same abstraction level as query plans. In comparison, a query formulated in CQL will need to be transformed by the query parser/optimizer into a query plan. The advantage of this box-and-arrow approach is its flexibility and that boxes-and-arrows are easy to understand as a query composition language. However, since Aurora/Borealis uses a set of operators different from the relational operators, many standard query optimization techniques may not be directly applicable. Certain automatic query rewrites are possible, but opportunities for query rewrite and optimization are limited in the presence of user-defined operators, due to lack of knowledge of the algebraic properties of these operators. Therefore, in many cases it is essentially up to the user to define an efficient query plan based on her expertise.

2.3.3 The Cayuga Query Algebra and Language

The stream query languages we introduce so far focus on expressing queries that continuously report some transformed state of the input streams, such as continuously computing a sliding-window average price of the input stock quote prices. There is another class of continuous queries, which detect *complex event patterns* composed of multiple input events correlated over time. For example, a complex event pattern on stock quotes may consist of two consecutive IBM stock quotes, such that the price of the second quote is above that of the first one. The research topic of processing the latter type of queries, complex event processing, or CEP for short, has been an active research area since the mid 1990s, and now with the emergence of RFID systems and Internet-scale monitoring applications, it is becoming increasingly practically relevant.

Although the stream query languages and systems we introduced previously are ex-

pressive enough for CEP, there are two inherent limitations in these approaches, due to the fact that these languages are not designed for CEP. First, they cannot *naturally* express patterns. As was mentioned earlier, to express an event pattern involving multiple events from the same input stream in CQL, a multi-way self-join on that stream is needed. Also, the temporal constraints among these events have to be explicitly specified in the WHERE clause of CQL, which makes it hard to recognize that such a CQL query is expressing a complex event pattern. Second, the query formulations of these complex event patterns in the above stream languages are likely to result in inefficient query execution plans. We will elaborate on this point in Section 2.4. One way of adding CEP functionality into the above stream languages is to extend them with primitives specially designed for CEP, such as the sequence operator introduced in Section 2.2.1

The Cayuga query algebra is specifically designed for large-scale CEP. It is inspired by regular expression operators, which are designed for matching patterns from an input sequence of letters drawn from a fixed alphabet. However, a data stream is richer than a sequence of letters, since each each event contains a set of attribute-value pairs, and the attribute values could be drawn from an infinite domain, instead of a finite alphabet. Therefore, Cayuga algebra also includes relational unary operators for filtering and transforming each individual input event based on its content. The resulting algebra is powerful and natural to express event patterns, and thanks to the well-developed automata theory, queries expressed in Cayuga algebra can be translated into finite state automata, where state sharing can be exploited for large scale event processing.

Here we briefly describe the Cayuga algebraic operators. We will present the details in the next chapter. Cayuga algebra contains operators drawn from relational algebra, such as σ, π and $\cup$. It also contains a unary aggregate α_f operator, which applies the

aggregate function f to its input stream. The power of this algebra for CEP comes from the two binary operators specially designed for expressing the building blocks of a complex event pattern. One of the two operators, the sequence operator, has been described in Section 2.2.2. The other operator is a generalization of the binary sequence, in that it is able to produce a complex event pattern involving unboundedly many input events, by iteratively concatenating input events with the pattern being built so far. These two operators bear certain resemblance to the two regular expression operators $\cdot$ and Kleene-*.

The query algebra is the formal foundation of Cayuga stream system. To allow users to interact with the system in a user friendly way, an SQL-style query language has been developed. It is essentially a syntactically-sugared version of the query algebra.

2.4 Stream Query Processing

So far we have described the elements that lay out the theoretical foundation of data stream processing – the timestamp model, and the query language. They serve as a formal contract between the user and the system on the behavior of the stream processing system; namely, *what* queries can be supported by the system, and *what* output the queries will produce. In this section, we will focus on *how* the processing system can evaluate the queries against continuously incoming streams, and produce results that conform to the query semantics.

Section 2.4.1 describes the evaluation techniques of typical stream operators, such as selection and window join. Since the evaluation algorithms of a *single* instance of these operators have been well studied in the relational query processing literature, and these techniques are directly applicable in the stream setting, the research focus here is

on how to efficiently evaluate a set of instances of these operators in a holistic fashion instead of sequentially. These techniques will serve as the building blocks in a stream query processing engine.

Section 2.4.2 focuses on some important scheduling issues of the operators in the query plans. It is well known that in relational query processing, the iterator interface [38] of query operators simplifies the control flow of the query engine. However, as we will describe in Section 2.4.2, the iterator interface is no longer suitable for stream query processing, which calls for new techniques of operator scheduling.

Section 2.4.3 presents techniques for distributed stream query processing, as well as techniques that make the system fault tolerant. This is motivated by the observation that in some streaming scenarios, the incoming data rate to the stream processing system may be high enough to exceed the processing power of a centralized system. Also, given the critical real-time constraints in some applications, the stream processing system cannot afford to keep its state in a persistent storage for crash recovery, as in the case for transaction processing. Techniques for achieving scalability beyond a single machine, as well as fault tolerance without decreasing the throughput of the stream processing are thus needed.

2.4.1 Evaluation Techniques of Stream Operators

Given a single instance of operator type $\mathcal{T}$, where $\mathcal{T}$ could be selection or some other type of operators, it is straightforward to process its input stream tuples efficiently. For example, for a single selection operator with selection predicate θ, we simply evaluate θ on each incoming stream tuple, and only output those tuples that satisfy θ. This evaluation technique on a single operator instance can be extended to evaluating a set of

operator instances in a naive way: for each incoming stream tuple, sequentially evaluate the set of operator instances. However, this naive method does not scale well with the number of operator instances, since it treats the evaluation of each operator instance as an independent computational task, and does not attempt to exploit the commonality among these tasks. The problem we try to address here is, *given a set of instances of operator type $\mathcal{T}$, we would like to efficiently evaluate them against each incoming stream tuple.* We refer to this problem as the *multi-operator processing* (or MOP for short) problem.

In this section, we focus on a few representative techniques for how to efficiently evaluate multiple instances of selection and window join operators respectively, as well as multiple complex event patterns. There has also been work on evaluating a set of window aggregates and group-by operators respectively [12, 99]. We refer the interested readers to these technical papers.

Multiple Selection Operators

For each incoming stream tuple, how to evaluate a set of selection operators on that tuple, such that the overall computational cost is significantly lower than the naive strategy of sequential evaluation? Let us first consider the extreme case of evaluating a set of N selection operators with identical predicates. In this workload, instead of evaluating the operators separately, resulting in repeated evaluation of the same predicate for N times, we could evaluate it once and obtain the result for all the N operators. Although this workload is rather unlikely in practice, it conveys an important idea of efficient MOP on selection operators, which allows us to generalize the above technique to handle other workloads of selection operators. This idea is to explore the *relationship* among these operators.

Next, consider an input stream tuple e containing an integer value v for attribute X in its payload. We have a set of selection operators, denoted as $\sigma_1, \cdots \sigma_N$, each specifying a single equality predicate comparing $e.X$ with a different constant value c_i for operator σ_i. Clearly, e satisfies σ_i iff $c_i = v$. We observe that the evaluation results of these selection operators with different constant values are mutually exclusive – if σ_i is satisfied by e for some i, then no other selection operators are satisfied also by e. Therefore, we could use a hash table to index these selection operators on the constant value c_i's. For the input stream tuple e, we use $e.X = v$ as the search key to probe this hash table. If we hit a bucket corresponding to key c_i, we know the selection operator σ_i is satisfied. Again, in this workload, we managed to evaluate the set of N selection operators in $O(1)$ time, assuming a good hash function is used for the hash table index. This technique can be further generalized to evaluate a set of selection operators where each operator specifies a conjunction of simple equality predicates. Also, it is possible to generalize this technique to handle inequality (range) predicates based on B+ trees.

Le Subscribe [33] brings the above idea even further, and proposes hashing based index structures to organize the set of input selection operators for efficient evaluation. As in the standard index selection problem for relational query evaluation, instantiating a new index incurs certain cost. Therefore, it becomes an optimization problem on how to select a set of indices, referred to as a configuration, such that the overall evaluation cost of the set of given predicates is minimized. Le Subscribe proposes a greedy algorithm for index selection, where in each step it weighs the benefit and cost of selecting a new index based on a cost model. In addition, it also addresses the problem of dynamically changing the index configuration, to deal with the case of predicates being inserted and removed at system run-time.

In short, the evaluation of multiple selection operators is a well studied problem in

the stream processing literature.

Multiple Window Joins

A window join operator reads tuples in the current window of its two input streams, and produce the joined tuples as output. The two input streams of the join operator could have different window specifications. However, for ease of exposition, in this section we assume they have the same window size. For example, the join operator on stream S_1 and S_2 with time-based window size T works as follows: For each input tuple S_1 with timestamp t, the tuple is joined with all the tuples from S_2 that have timestamp no earlier than $t - T$. Each incoming tuple from S_2 is processed similarly. The semantics of tuple-based window join is defined similarly.

Given a set of window join operators with the same input streams and join predicates but different specification for window lengths, which we denoted as $j_1, \cdots, j_k$, how to process them efficiently? As was observed early in the stream processing literature, if the goal is to minimize the total run time cost, then the join processing should be *aligned* with the join operator with the largest window size. That is, we draw the input tuples of join from tuples in the largest window, evaluate the join predicate on them, and for each output tuple t that satisfies the join predicate, we put it on the output stream of $j_i (1 \leq i \leq k)$ iff the two input tuples that form t are within the window specification of j_i.

However, if the optimization metric is something other than to minimize the total run time, then the above query evaluation strategy may not be optimal. [43] proposes two alternative metrics, as well as corresponding join evaluation techniques to optimize these metrics. The first one, Smallest Window Only (SWT), favors those joins with

small window sizes by producing query output for them first. The second one, Maximum Query Throughput, chooses to process those tuples that could serve the maximum number of window joins per unit time.

Multiple Complex Event Patterns

By exploiting the relationship of the query algebra to automata based query execution, the Cayuga event processing engine can efficiently process a large number of complex event patterns. Specifically, each query is first translated into a set of automata. These automata are then "merged" with existing ones in the engine. During the merging process, two optimization techniques are used. First, two automata with the same prefix of states can merge these states, thus achieving sharing of computation and storage. Second, the *filter predicates* (specified in the θ parameter of $;_\theta$ introduced in section 2.2.2) are indexed, such that for each incoming event, instead of evaluating these filter predicates sequentially, the engine can probe the appropriate predicate indices and retrieve the evaluation results more efficiently. These two techniques are the keys to scalable event processing in Cayuga.

2.4.2 Operator Scheduling

As was mentioned earlier, in a relational query engine, the iterator interface significantly simplifies the control flow of the query engine. To produce output tuples for a query plan, the top level operator of that plan "pulls" tuples from its children by invoking the GetNext method on these child operators. Each child then in turn pulls tuples from its children in order to produce output tuples. This way, the control flow passes from the top level operator to leaf operators in the query plan, and no explicit operator scheduling

logic is required. However, for stream query processing, if iterator interface is used, some leaf operator that pulls tuples from its input stream may have to be blocked, if at that moment the input stream does not have any tuple that has arrived at the system but has not been processed. This is undesirable because there might be tuples from other input streams waiting to be processed by the system.

The opposite of pull-based operator scheduling is push-based. In push-based scheduling, the system keeps a global buffer of incoming events from all the input streams. The buffer then feeds the query engine one event at a time. This input event "flows" through the relevant operators in the query plan in a bottom-up fashion, and the system does not start processing the next event until it finishes processing the current tuple; e.g., when that tuple (or some transformation of it) is output by a top-level operator of the query plan, or when it is filtered out by a selection operator, or when it (or some transformation of it) is stored in an operator state. This push-based operator scheduling has the advantage of never having to block the query processing unnecessarily. This scheduling is adopted by Cayuga.

In the above two scheduling strategies, the control flow of the system is tightly coupled with the data flow. In the pull-based scheduling, it is the parent that "drives" its children, whereas the situation is the opposite in the push-based scheduling. It is possible to decouple control flow from data flow, by adding queues between operators. STREAM takes this approach to an extreme, by associating a queue with each operator for buffering input tuples. This achieves maximal flexibility in operator scheduling. That is, at any time instance, the operator scheduler could choose an arbitrary operator in the system to evaluate for a period of time, and then switch to evaluating another operator. However, this additional flexibility is accompanied with extra cost in running the scheduling logic. Also, these input queues consume extra memory resource. It is

worth noting that by associating with each operator an input queue, a system such as STREAM can be easily extended for parallel query evaluation, where operators can run concurrently in different threads. There has also been some work on developing a hybrid push/pull based system, such as the FJORD operator from TelegraphCQ [62]. Under this framework, a query plan could involve some operators that are push based, and others that are pull based.

Finally, we describe an important idea called *eddy* for operator scheduling in stream query processing [13]. Eddy opens up a new dimension in the solution space of operator scheduling. The scheduling strategies presented so far all fix the data flow of input tuples, and differ in the control flow aspect. In comparison, eddy is a data flow operator which dynamically decides for each incoming tuple, in which order it should *route* the tuple to the set of operators in the query plan. After an operator receives a tuple from eddy and evaluates it, it records lineage information explicitly within the tuple, and sends it back to eddy. Such lineage information indicates which set of operators have been evaluated on this tuple, as well as the evaluation results (for predicate evaluation). The information is essential for eddy to make routing decisions. Eddy is useful in the setting where there are multiple candidate data flows for input tuples. Queries involving selection and join operators provide such a setting, due to the commutativity of these operators. Since each tuple can go through a different sequence of operator evaluation, the eddy-based query evaluation is the most flexible one in adapting to the potential change of the stream workload. A disadvantage of eddy is in the potentially large lineage information that has to be recorded for each tuple. In the settings where the stream workloads do not vary frequently, the level of flexibility in scheduling promised by eddy might become overkill, and negatively impact the system performance. However, it is possible to remedy this situation by mixing eddy-based scheduling and traditional operator scheduling. For example, instead of recording lineage information for each

single tuple, the eddy-based system could track lineage for each group of n tuples, such that the data flow of the operators within the same group is fixed. By tuning the parameter n given a query and stream workload, it is possible to achieve a nice balance between the benefit brought by the flexibility of scheduling and the cost of achieving such flexibility. It is also possible to dynamically change the parameter n to adapt to the changing workload.

2.4.3 Scalability and Fault Tolerance

When the incoming data rate is higher than one machine could handle, new techniques on disributed query processing need to be developed to extend stream processing to a set of machines, achieving higher scalability. In the setting of distributed query processing, the notion of a query plan is generalized to mappings of query operators to machine hosts. *Load balancing* is an essential element in a distributed stream processing system. Load balancing techniques take as input a particular workload, and construct a query plan that minimizes load variance and maximizes load correlation among machine hosts. When the stream workload changes in system run-time, such that the load of one machine exceeds its capacity, the load-balancing techniques manage to dynamically migrate some of the load to other machines. [97, 79]. Also, in the scenario where it is expensive or impossible to migrate loads at run-time, researchers have developed techniques to construct a query plan which is resilient to load variations [96].

When a stream processing system is overloaded, another approach is to do *load shedding*, where the system chooses a set of input tuples to drop in order for it to keep up with the incoming data rate, and at the same time either minimize the impact on query output, or conform to a given QoS specification. A number of techniques have been

developed to do load shedding either for the entire system or for individual operators [15, 16, 88].

Finally, without techniques for adding fault tolerance to the stream processing system, the query processing state may be lost when the system crashes. A variant of the process-pair approach [39] can be used for achieving fault tolerance in the stream processing setting [80, 48]. The basic idea is to pair up a secondary machine with each stream processing server, referred to as the primary machine. Both machines read the same input streams, evaluate the same operators, but the secondary machine does not emit any output in the normal case. When the primary machine dies, the secondary machine takes over and continues the stream processing. Among all the crash recovery techniques to be described here, process-pair is the most efficient. Its major disadvantage is a relatively inefficient utilization of resources. [48] identified the inherent trade-off between the recovery cost and the amount of resources allocated for crash recovery, and proposes other recovery techniques that make different trade-offs between the above two factors. More recently, [17] investigates the setting where network links could fail, and [49] proposes more efficient recovery schemes, which use resources from multiple machines to speed up the recovery process.

CHAPTER 3

CAYUGA – THE EVENT MODEL, ALGEBRA, AND PROCESSING MODEL

3.1 Introduction

Publish/Subscribe is a popular paradigm for users to express their interests ("subscriptions") in certain kinds of events ("publications"). Traditional publish/subscribe (pub/sub) systems such as topic-based and content-based pub/sub systems allow users to express stateless subscriptions that are evaluated over each event that arrives at the system; and there has been much work on efficient implementations [33]. However, many applications require the ability to handle *stateful* subscriptions, which involve more than a single event. Users want to be notified with customized witness events as soon one of their stateful subscriptions is satisfied.

To understand what we mean by a stateful subscription, we present two example applications that motivate the types of subscriptions we would like to handle.

Example 1: Stock Ticker Event Monitoring. Consider a system that permits financial analysts to compose subscriptions over a stream of stock ticks [1]. A traditional subscription this system would be one that looks at a single stock quote and evaluates it only according to the values within this quote, as in the following example.

Example 1 *Forward all IBM quotes for which the price is above $100.*

However, an important feature of event processing systems is the ability to detect specific *sequences* of events. To detect sequences, the system has to maintain *state* about events that have previously entered the system. For example, consider the following subscription.

Example 2 *Forward all IBM quotes for which the price has just crossed above $100.*

In this quote, the system has to "remember" whether the previous IBM quote was below $100, as we disregard future quotes once we reach our target price, unless IBM dips below $100 again. In other words, we are looking for all pairs of successive IBM quotes where the first quote is below $100 and the second quote is above.

The previous examples compared the stock quotes to constants. For more sophisticated queries, we need *parameterized* subscriptions, where the events are filtered according to parameters bound at run-time to values seen in earlier events. In each of the examples below, instead of having to register a subscription for each possible stock symbol, we register a single subscription with a *parameter* for the stock name that is set at run time.

Example 3 *Forward any stock quote whose price p is 5% greater than that of the previous quote of that stock.*

Example 4 *Forward the quotes for any stock that has been increasing monotonically in price for at least 30 minutes.*

Finally, subscriptions such as Example 4 can return arbitrarily long sequences of stock quotes. For these types of subscriptions, we may want to summarize the information using *aggregation*. The following subscription uses aggregation to compute the average of all of the stock quotes from the last 52 weeks.

Example 5 *Forward all IBM stock quotes that are above the 52-week average.*

Example 2: RSS Feed Monitoring. Our second motivating application is online RSS Feed Message Brokering. RSS feeds have become increasingly important for online exchange of news and opinions. With a stateful pub/sub system, users can monitor RSS Feeds and register complex subscriptions that notify the users as soon as their requested RSS message sequences have occured.

While the domain of RSS monitoring is different from stock feeds, our desired subscription types are still very much the same. The following two queries illustrate both sequencing and parameterization.

Example 6 *Once Apple posts a product announcement on its web page, send me the first post referencing (i.e., containing a link to) this article from any of the blogs to which I subscribe.*

Example 7 *Whenever a new Apple rumor is posted on ThinkSecret.com, send me any posting which either references this rumor directly, or which references a previous posting forwarded by this subscription.*

All of these example queries can be processed by Data Stream Management Systems (DSMS)[3, 67, 24]. These systems have powerful query languages that allow them to express joins and aggregation over streams. However, these systems are very complex and have limited scalability with respect to the number of subscriptions. Traditional pub/sub systems, on the other hand, scale to millions of registered subscriptions and very high event rates, but have limited expressive power. In these systems, users can only submit subscriptions that are predicates to be evaluated on single events. Any operation across multiple events must be handled externally. Our proposed stateful pub/sub system, is an attempt at a middle ground between these two extremes, as illustrated in Table 3.1. Our

system can support subscriptions spanning multiple events, involving parameterization and aggregation, while maintaining scalability in the number of subscriptions and event rate.

Table 3.1: Trade-Offs between Pub/Sub and Data Stream Management Systems

Complexity of Subscriptions	Number of Concurrent Subscriptions: few	Number of Concurrent Subscriptions: many
low	uninteresting	Pub/Sub
high	DSMS	Stateful Pub/Sub

Our design of a stateful pub/sub system is very closely related to work on *event systems*. Event systems can have query languages (called *event algebras*) that can compose complex events from either basic or complex events arriving online. However, we have observed an unfortunate dichotomy between theoretical and systems-oriented approaches in this area. Theoretical approaches, based on formal languages and well-defined semantics, generally lack efficient, scalable implementations. Systems approaches usually lack a precise formal specification, limiting the opportunities for query optimization and query rewrites. Indeed, previous work has shown that the lack of clean operator semantics can lead to unexpected and undesirable behavior of complex algebra expressions [35, 100]. Our approach was informed by this dichotomy, and we have taken great care to define a language that can express very powerful subscriptions, has a precise formal semantics, and can be implemented efficiently.

Our Contributions. In this chapter, we propose Cayuga, a stateful publish/subscribe system based on a nondeterministic finite state automata (NFA) model. We start in Section 3.2 by introducing the Cayuga event algebra, which can express all example subscriptions listed in Section 3.1. We illustrate how algebra expressions map to lin-

ear finite state automata with self-loops and buffers (Section 3.2). To the best of our knowledge, this is the first work that combines a formal event language definition with a methodology to efficiently implement the language. We conclude this chapter in Section 3.4, and present the design and implementation challenges of the Cayuga system in the next chapter. We also defer the discussion on related work until next chapter.

In closing this introduction, we would like to emphasize two important aspects of our approach. First, instead of adding features to a pub/sub system in an ad-hoc fashion, our system is based on formal language operators and therefore provides unambiguous query semantics that are necessary for query optimization. Second, compared to similar approaches that use NFAs for scalability such as YFilter [31], Cayuga supports powerful language features such as parameterization and aggregation that require us to extend beyond prior work. One interesting result from our experimental study is that common optimization techniques used in NFA-based systems, such as state merging, have only limited effectiveness for the workloads that we consider. On the other hand, some of our novel MQO techniques could potentially be applied to other NFA-based systems.

3.2 Cayuga Event Algebra

To ensure that our systems has clean operator semantics, we introduce a formal event algebra. Our event algebra consists of a data model for event streams plus operators for producing new streams from existing ones. This way, the output streams of any algebra (sub)expressions can be fed directly into other operators as input streams. We call those events produced by our algebra (sub)expressions *composite events* to distinguish them from *primitive events* coming from external data sources.

An event stream, denoted as S, is an infinite sequence of event tuples

$\langle \overline{a}; \text{START}, \text{END} \rangle$. As in the relational model, $\overline{a} = (a_1, \ldots, a_n)$ are data values with corresponding attributes (symbolic names). We assume each event stream has a fixed schema. Given an event e and attribute **x**, we let $e.\mathbf{x}$ represent the value of e for that attribute.

START and END are temporal values representing the start and end timestamps of the event, respectively. We assume events (in the same stream or multiple streams) arrive at the system in the ascending order of their END values, and they are processed in the same order. That is, event e_1 is processed before e_2 if $e_1.\text{END} \leq e_2.\text{END}$. However, we do allow for simultaneous events (i.e. events with identical END values), and make no guarantees on the order of such events. While start time does not effect the order of the events in the stream, we store it to avoid well-known problems involving concatenation of complex events [35]. In particular, the stream may contain events with non-zero duration as well as overlapping events (events with overlapping time stamp intervals).

In this chapter, we have two representations for timestamps: wall clock representation such as $9:10$, and integer representation. The latter is chosen for ease of exposition in some of our later examples. For example, in the stock monitoring application, assume the stream of stock sales published by the data source has fields $\langle \texttt{Name}, \texttt{Price}, \texttt{Vol}; \text{START}, \text{END} \rangle$. A sample event from that stream then could be the tuple $\langle \text{IBM}, 85, 15000; 9{:}10, 9{:}10 \rangle$, representing a transaction on IBM stock at price \$85 with volume 15000 shares at time 9:10AM[1]. This event has a trivial duration; i.e., the start and end timestamps are identical, because each sale is an instantaneous event. However, an event can also have a non-trivial duration. For example, $\langle \text{IBM}, 85, 90; 9{:}10, 9{:}15 \rangle$ from a stream with data attributes `Name`, `oldPrice`, and `newPrice` could indicate that the price of IBM increased from \$85 to \$90 between 9:10AM and 9:15AM.

[1]For simplicity we omitted the representation for AM or PM in the event timestamp fields as well its exact date.

To this data model, we add an operator algebra that takes streams as input and produces streams as output. Our algebra is similar to the relational algebra, except that join has been replaced by operators that add the expressive power of regular expressions. As an abuse of notation, we will sometimes apply an algebra operator to a single event; the meaning of such an expression is the operator applied to a stream containing only that single event. In the sections below we define each of our operators.

3.2.1 Unary Operators

The operators introduced in this section are well known from the relational algebra. The first unary operator is the standard **projection** operator $\pi_{\mathbb{X}}$, where $\mathbb{X}$ is a set of attributes. As the binary operators depend on the timestamp values, we cannot allow users to project out timestamps. Thus projection can only affect data values.

The operator σ_θ is the standard **selection** operator σ_θ, where θ is any selection formula. As with the relational algebra, a selection formula can be any boolean combination of atomic predicates of the form τ_1 relop τ_2, where the τ_i are arithmetic combinations of attributes, constants and user defined functions, and relop is one of $=, \neq, \leq, <, \geq, >$.

For technical reasons, we do not allow users to query timestamps directly, particularly in arithmetic expressions. However, we do allow selection constraints on the *duration* of an event, defined as END $-$ START (we treat time as discrete, so the duration of an event is the number of clock ticks it spans). Thus we allow selection formulas to contain predicate DUR relop c where DUR represents the event duration, c is a constant, and relop is as above.

We write $e \models \theta$ when selection formula θ is true of event e. So for any stream S,

$$\sigma_\theta(S) = \{ e \in S \mid e \models \theta \}$$

For example, consider Example 1 in Section 3.1. If S is our stock stream, then we can express this subscription as $\sigma_{\texttt{Name}=\text{IBM}\wedge\texttt{Price}>\$100}(S)$. Any atomic (i.e. no conjunction or disjunction) selection predicate that refers to an attribute not in S evaluates to FALSE. Hence $\sigma_{\texttt{Name}=\text{IBM}\wedge\texttt{color}>\text{blue}}(S)$ returns the empty set, while $\sigma_{\texttt{Name}=\text{IBM}\vee\texttt{color}>\text{blue}}(S)$ returns all IBM stock quotes. While we could simply say that $\sigma_\theta(S)$ is undefined when θ has attributes not in S, this definition simplifies the proof in Section 3.3.3.

The last unary operator is the **renaming** operator ρ_f where f is a one-to-one function from one set of attributes to another set of attributes. For example, if S is our stock stream from Section 3.2, and f maps $\texttt{Price} \mapsto \texttt{oldprice}$, then the stream $\rho_f(S)$ has schema $\langle \texttt{Name}, \texttt{oldPrice}, \texttt{Vol}; \text{START}, \text{END} \rangle$. It is important to note that a renaming function applied to a stream simply changes the names of the *data attributes* in the stream schema; the timestamps cannot be renamed.

As the renaming operator affects only the schema of a stream and not its contents, we will often ignore this operator for ease of exposition. Instead, we usually index the attributes of an event by the ID of the input stream, making renaming implicit. For example, the `Name` attribute of events from stream S_1 will be referred to as $S_1.\texttt{Name}$. From here on, we will only use an explicit renaming operator when we want to formally specify a certain schema.

3.2.2 Union and Sequencing

The unary operators enable filtering and transformation of individual events based on their attribute values, and therefore are equivalent the expressive power of a standard

content-based pub/sub system. The added expressive power of our algebra lies in the binary operators, which support subscriptions over multiple events. All of these operators are motivated by a corresponding operator in regular expressions.

The first binary operator is the standard **union** operator $\cup$, where $S_1 \cup S_2$ is defined as $\{\, e \mid e \in S_1 \text{ or } e \in S_2 \,\}$. As in the relational model, we require "union compatible" schemas on the input streams S_1 and S_2, which is achievable by the renaming operator ρ_f.

In addition to the union operator, we also introduce a **sequencing** operator. Before we define this operator on streams, we first define it on individual events. Let e_1, e_2 be two *non-overlapping* events (i.e. $e_1.t_1 < e_2.t_0$). We define the sequenced event $e_1 ; e_2 = \langle \overline{c}; e_1.t_0, e_2.t_1 \rangle$ where $\overline{c}$ is the concatenation of the data values from e_1 and e_2, provided that the attributes of the two events are distinct. In the case where a data attribute is present in both e_1 and e_2, $\overline{c}$ will only contain the value from e_2 for this attribute. For example, let $e_1 = \langle \text{IBM}, 85, 1000; 1, 1 \rangle$ have data schema – that is, not including the timestamps – $(\texttt{Name}, \texttt{Price}, \texttt{Vol})$ and $e_2 = \langle \text{MSFT}, 27; 2, 3 \rangle$ have data schema $(\texttt{company}, \texttt{Price})$. Sequencing these, we get the event $e_1 ; e_2 = \langle \text{IBM}, 27, 1000, \text{MSFT}; 1, 3 \rangle$ with data schema $(\texttt{Name}, \texttt{Price}, \texttt{Vol}, \texttt{company})$.

We extend this definition to two streams S_1 and S_2 by sequencing any event from S_1 with the *next* event from S_2.

$$S_1 ; S_2 = \{ e_1 ; e_2 \mid e_1 \in S_1, e_2 \in S_2, \text{ and } \nexists e_2' \in S_2 \text{ such that } e_2'.\text{END} < e_2.\text{END} \}$$

This operator allows us to express Example 2 as the expression

$$\sigma_{S_1.price \leq \$100 \wedge S_2.price > \$100}(\sigma_{S_1.name = \text{IBM}}(S_1) ; \sigma_{S_2.name = \text{IBM}}(S_2))$$

where both S_1 and S_2 refer to the base stream of stock quotes (think of S_1 and S_2 as having the renaming operator applied to the base stream to distinguish the attributes of

the same name, such as S_1.Name and S_2.Name).

As an extended example of sequencing, let S_1 and S_2 be streams as shown in Figure 3.1, for which we want to compute $S_1 ; S_2$. The result of this subscription is a single composite event, which contains the data fields of e_1 and e_4, and whose start and end timestamps are the start time of e_1 and the end time of e_4, respectively. Event e_2 cannot be sequenced with e_1, because their time intervals overlap. Furthermore, although e_3 can be sequenced with e_1, it is e_4 that is be picked by ;, since starts after e_1 ends, and ends before e_3 ends (see the definition of sequencing above). For the same reason, any other event ending after e_4 cannot be sequenced with e_1 either.

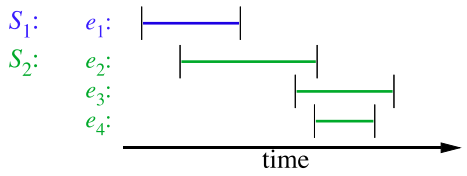


Figure 3.1: Time intervals of arriving events

Sometimes we do not want the absolute next event in a stream, but would rather have the next event satisfying some condition with a run-time parameter. Recall that Example 3 pairs together stock quotes where the name for the second stock quote is conditional on the name from the first quote. For this we need **conditional sequencing**. For streams S_1 and S_2, and selection formula θ, we define

$$S_1 \underset{\theta}{;} S_2 = \{e_1 ; e_2 \models \theta \mid e_i \in S_i \text{ and } \nexists e_2' \in S_2 \text{ with } e_2'.\text{END} < e_2.\text{END}, e_1 ; e_2' \models \theta\}$$

Intuitively, this operator computes sequences of consecutive events, just like regular sequencing, except that it filters out those events from S_2 that do not satisfy θ. Given this operator, we implement Example 3 as

$$\sigma_{1.05*S_1.\texttt{Price}<=S_2.\texttt{Price}}\Big(\underset{S_1.name=S_2.name}{S_1 \; ; \; S_2}\Big)$$

where both S_1 and S_2 refer to the base stream of stock quotes.

Understanding Sequencing

While sequencing is present in existing event systems like Snoop [22], our use of interval timestamps makes our definition of sequencing unique. Furthermore, conditional sequencing is unique to the Cayuga event algebra. There a several important design decisions that have gone into the formulation of these operators. For the interested reader, we expand upon these decisions here.

First of all, even though other event systems implement some form of sequencing, Figure 3.1 demonstrates that interval timestamps makes the Cayuga implementation differ in two important ways from existing systems such as Snoop [22]. First of all, two events $e_1 \in S_1$ and $e_2 \in S_2$ can only be sequenced if e_2 starts after e_1 has ended. Without this requirement, it can be shown that $S_1 ; (S_2 ; S_3)$ is equivalent to $S_2 ; (S_1 ; S_3)$ [35]. This result is a confusing and unexpected query semantics for sequencing. Notice, however, that if such semantics was actually desired, our algebra could easily be modified to support it.

Second, out of all events from S_2 that start after $e_1 \in S_1$ has ended, the operator selects the one with the earliest *end* timestamp. In the above example, e_3 is not combined with e_1, even though it started before e_4. This choice of operator semantics is natural, because an event *occurs* (is detected by the system) at its end time and hence the event order is determined by the end timestamp. From an implementation point of view this semantics is desirable as well, because before the end time of an event, the system has no knowledge if this event will ever occur or not.

Finally, in regards to conditional sequencing ($;_\theta$), note that normal sequencing ($;$) is

just the special case where $\theta = \text{TRUE}$. However, the reverse is not possible; we cannot define conditional sequencing from normal sequencing and selection. Consider again Example 3. We need to pair two successive quotes of the same stock. We cannot select stock pairs after sequencing. $\sigma_{S_1.\texttt{Name}=S_2.\texttt{Name}}(S_1 \,;\, S_2)$ first pairs successive stock quotes, and then selects pairs of the same name; this will return nothing if our stock ticker has several different stocks with quotes interleaved. Therefore, we need to filter the stocks by name before sequencing them, like we did in Example 2. However, we cannot do this as the stock name is now a run-time parameter, not a constant. In conditional sequencing, the filter θ works as "group-by", allowing us to group stock quotes by name. Thus our fundamental operation is conditional sequencing, not sequencing. Hence, from here on, we let ; simply be shorthand notation for the conditional operator $;_{\text{TRUE}}$.

3.2.3 Iteration

The last binary operator is motivated by the Kleene-+ operator. For example, suppose we want to detect an upward trend in a stock price as was shown in Example 4. To express this type of subscription, we introduce the **iteration** operator $\mu_{\mathfrak{F},\theta}(S_1, S_2)$. Informally, we can think of $\mu_{\mathfrak{F},\theta}(S_1, S_2)$ as a repeated application of $;_\theta$ and unary expression $\mathfrak{F}$ for each successful concatenation via $;_\theta$. In other words, it is the set of events

$$\mathfrak{F}(S_1 \underset{\theta}{;} S_2) \;\cup\; \mathfrak{F}(\mathfrak{F}(S_1 \underset{\theta}{;} S_2) \underset{\theta}{;} S_2) \;\cup\; \mathfrak{F}(\mathfrak{F}(\mathfrak{F}(S_1 \underset{\theta}{;} S_2) \underset{\theta}{;} S_2) \underset{\theta}{;} S_2) \;\cup\; \cdots$$

Each clause separated by the $\cup$ operator corresponds to an iteration of processing an event from S_2 which satisfies θ. The additional parameter $\mathfrak{F}$, a composition of unary operators, enables us to modify the result of each iteration. Thus μ acts as a fixed point operator, applying the operator $;_\theta$ on each incoming event repeatedly until it produces an empty result (when the incoming event violates the predicate constraint expressed in $\mathfrak{F}$).

However, this informal definition is not quite right for two reasons. First of all, because Cayuga is intended to be highly scalable, we want to avoid unbounded storage. Furthermore, to allow later subscriptions to query the output of earlier subscriptions, the output of any subscription should have a fixed, well-defined schema. To meet both of these requirements, the output schema of μ will only contain the attribute values from stream S_1 and the values from the most recent iteration of S_2. For any attribute ATT common in S_1 and S_2, we refer to the value from the S_1 event in the output of μ via ATT.`first`. For any of any ATT of S_2, we have two corresponding attributes in the computation of μ, ATT and ATT.`last`. ATT refers the the value from the final iteration of μ, and is present in the output schema of μ. ATT.`last`, on the other hand, refers the previous iteration of μ, and is initially the corresponding attribute value in S_1. Attributes of form ATT.`last` will be overwritten by each iteration and are only present during the internal computation of μ – they are not exposed in the output schema of μ.

Formally, we define this $\mu_{\mathfrak{F},\theta}(S_1, S_2)$ as follows. First, we require the data schema of S_2 be a subset of the schema of S_1. This requirement can be fulfilled with renaming and projection operator applied to S_2. We then set $\mu_{\mathfrak{F},\theta}(S_1, S_2) = \bigcup_{n\geq 1} \mathcal{I}^{[n]}$ where

$$\begin{aligned} \mathcal{I}^{[0]} &= \left\{ \left\langle \overline{a}\overline{b}; t_0, t_1 \right\rangle \;\middle|\; \langle \overline{c}; t_0, t_1\rangle \in S_1, \overline{a} = \rho_{f_1}(\overline{c}), \overline{b} = \pi_{\mathbb{S}_2}(\overline{c}) \right\} \\ \mathcal{I}^{[n+1]} &= \pi_{\overline{f_2(\mathbb{S}_2)}} \circ \mathfrak{F}\big(\rho_{f_2}(\mathcal{I}^{[n]}) \underset{\theta}{;} S_2\big). \end{aligned} \tag{3.1}$$

where $\mathbb{S}_i$ is the data schema of the stream S_i for $i = 1, 2$. Here $\circ$ is the standard composition operator; that is, for two operators ω_1, ω_2, and input x, the expression $\omega_1 \circ \omega_2(x)$ is equivalent to $\omega_1(\omega_2(x))$. We use this notation to improve readability.

In the definition of $\mathcal{I}^{[0]}$, $\overline{a}$ represents data attribute values in S_1, and renaming function f_1 renames each data attribute ATT common in stream S_1 and S_2 to ATT.`first`. Similarly, f_2 renames each data attribute ATT in stream S_2 to ATT.`last`. In the definition of the $\mathcal{I}^{[n+1]}$, data attributes ATT of $\mathcal{I}^{[n]}$ are first renamed to ATT.`last` by ρ_{f_2}, and

then $\mathfrak{F}$ is applied to the concatenation of $\mathcal{I}^{[n]}$ and an S_2 event via $;_\theta$, followed by projection operator that preserves all attributes in the schema of $\mathfrak{F}$ other than those temporary attributes ATT.`last`. We chose the notation $\pi_{\overline{f_2(\mathbb{S}_2)}}$ to resent the complement of schema $f_2(\mathbb{S}_2)$ w.r.t. the schema of $\mathfrak{F}$.

As a syntax shortcut, another way to refer to attribute ATT.`first` in the output of $\mu_{\mathfrak{F},\theta}(S_1, S_2)$ is S_1.ATT. Similarly, ATT in the output of $\mu_{\mathfrak{F},\theta}(S_1, S_2)$ can be referred to as S_2.ATT as well. Notice such shortcuts can only be used to refer to the attributes in the output of μ – they carry different meanings when they are used in $\mathfrak{F}$ and θ.

With iteration, we can express Example 4 as

$$\sigma_{\text{DUR}\geq 30\text{ min}}\left(\mu_{\sigma_{S_2.\texttt{Price}>S_2.\texttt{Price.last}},S_1.\texttt{Name}=S_2.\texttt{Name}}(S_1, S_2)\right) \tag{3.2}$$

Like conditional sequencing and Example 3, $S_1.\texttt{Name} = S_2.\texttt{Name}$ works as a group-by operation; we are iterating over $;_{S_1.\texttt{Name}=S_2.\texttt{Name}}$. On the other hand, the unary operator $\sigma_{S_2.\texttt{Price}>S_2.\texttt{Price.last}}$ acts as a stopping condition, letting us know when to cease the iteration. Because μ is the union of all the iterations, the subscription $\mu_{\sigma_{S_2.\texttt{Price}>S_2.\texttt{Price.last}},S_1.\texttt{Name}=S_2.\texttt{Name}}(S_1, S_2)$ outputs all monotonic runs, no matter how small. Thus the exterior selection formula is used to select all those stock runs that have a duration of at least 30 minutes.

Understanding Iteration

The μ iteration operator is another operator that is unique to the Cayuga event algebra. It is also the operator responsible for the most confusion. Therefore, in this section we will try to clarify the definition of the μ operator with an extended example, as well as a discussion of the important design decisions that went into this operator. This section is only intended for those readers interested in understanding the μ operator, and is not

necessary for understanding the rest of the chapter.

First, let us illustrate the functionality of the μ operator in processing the formulation of Example 4 in (3.2). Consider the example stock stream

$$S_1 = S_2 = \left\{ \begin{array}{l} \langle \text{IBM}, 80; 1, 1\rangle, \langle \text{Dell}, 22; 2, 2\rangle, \langle \text{IBM}, 82; 3, 3\rangle, \\ \langle \text{Dell}, 24; 4, 4\rangle, \langle \text{IBM}, 84; 5, 5\rangle, \langle \text{Dell}, 22; 6, 6\rangle \end{array} \right\}$$

Note that for the sake of readability we have simplified the schema of the stream by removing the volume attribute, which is irrelevant for the subscription.

The initial set $\mathcal{I}^{[0]}$ is computed by duplicating the relevant attributes $S_1.\text{ATT}$ and renaming one copy of each to $\text{ATT.}\texttt{first}$. Hence the resulting schema of $\mathcal{I}^{[0]}$ is $\langle \texttt{Name.first}, \texttt{Price.first}, \texttt{Name}, \texttt{Price}; t_0, t_1\rangle$. So from our stream we obtain

$$\mathcal{I}^{[0]} = \left\{ \begin{array}{l} \langle \text{IBM}, 80, \text{IBM}, 80; 1, 1\rangle, \langle \text{Dell}, 22, \text{Dell}, 22; 2, 2\rangle, \langle \text{IBM}, 82, \text{IBM}, 82; 3, 3\rangle, \\ \langle \text{Dell}, 24, \text{Dell}, 24; 4, 4\rangle, \langle \text{IBM}, 84, \text{IBM}, 84; 5, 5\rangle, \langle \text{Dell}, 22, \text{Dell}, 22; 6, 6\rangle \end{array} \right\}$$

From this set, the first iteration $\mathcal{I}^{[1]}$ will hold all pairs of adjacent quotes of the same stock (θ_1 filters out quotes from other companies), in which the second quote is higher. The latter is enforced by θ_2, which removes all pairs of quotes with non-increasing price. This iteration has the same schema as the previous one. To achieve this, each attribute ATT from the result of the previous iteration is renamed to $\text{ATT.}\texttt{last}$ to preserve the attribute values of the S_2 event in the most recent iteration, and then attributes of form $\text{ATT.}\texttt{last}$ are projected out. This gives us

$$\begin{aligned} \mathcal{I}^{[1]} &= \pi_{\overline{f_2(\mathbb{S}_2)}} \circ \sigma_{S_2.\texttt{Price} > S_2.\texttt{Price.last}} \Big(\rho_{f_2}(\mathcal{I}^{[0]}) \underset{S_1.\texttt{Name}=S_2.\texttt{Name}}{;} S_2 \Big) \\ &= \{ \langle \text{IBM}, 80, \text{IBM}, 82; 1, 3\rangle, \langle \text{Dell}, 22, \text{Dell}, 24; 2, 4\rangle, \langle \text{IBM}, 82, \text{IBM}, 84; 3, 5\rangle \} \end{aligned}$$

Similarly, the next iteration is computed as

$$\begin{aligned} \mathcal{I}^{[2]} &= \pi_{\overline{f_2(\mathbb{S}_2)}} \circ \sigma_{S_2.\texttt{Price} > S_2.\texttt{Price.last}} \Big(\rho_{f_2}(\mathcal{I}^{[1]}) \underset{S_1.\texttt{Name}=S_2.\texttt{Name}}{;} S_2 \Big) \\ &= \{ \langle \text{IBM}, 80, \text{IBM}, 84; 1, 5\rangle \} \end{aligned}$$

After this point, $\mathcal{I}^{[n]}$ is empty for all $n > 2$. The result of the μ operator is then the union $\bigcup_{n \geq 1} \mathcal{I}^{[n]} = \mathcal{I}^{[1]} \cup \mathcal{I}^{[2]}$. Note that $\mathcal{I}^{[0]}$ is not included in the final output.

At first it might seem surprising that our algebra needs an operator as complex as $\mu_{\mathfrak{F},\theta}(S_1, S_2)$ to express the equivalent of something as simple as $(S_2)^+$ in regular languages. As with $;_\theta$, we would like some way of filtering out irrelevant events from S_2; this is the purpose of the θ in $\mu_{\mathfrak{F},\theta}$. In the above example, it was used to make sure that no Dell stock would be selected for a sequence of IBM prices, and vice versa. Placing a selection formula in $\mathfrak{F}$ also allows us to remove irrelevant events. However, unlike θ, we stop the iterating over events when the selection from $\mathfrak{F}$ fails, instead of just skipping over them. Thus we need two subscripts to distinguish the two different types of selection.

However, this explanation does not demonstrate why we chose $\mathfrak{F}$ for the first subscript instead of μ_{θ_1,θ_2} where θ_2 identifies the events to skip over and θ_1 identifies when to stop the iteration. As we shall see in Section 3.2.4, we generally want to combine aggregation with iteration, and the obvious way to implement aggregation is by applying an aggregate function at each step of the iteration. So we have generalized the operator to take an arbitrary unary operation $\mathfrak{F}$, and not just selection, as the iteration function.

Finally, there is the issue of why we have chosen for μ to be a binary operator, when Kleene-+ is unary. Naively, we could have defined a unary operator $(S)^\star_{\mathfrak{F},\theta}$ using much the same definition of μ. There are two important reasons for making μ binary. First of all, we need some way to initialize our attributes ATT.`last` so that we can use them in a subscription like Example 4. It is unclear what $(S)^\star_{\sigma_{S.\texttt{Price}>S.\texttt{Price.last}},\textsc{True}}$ if there is no stock quote to begin with.

More importantly, we want μ to be binary so that the stopping condition expressed

in $\mathfrak{F}$ can depend upon earlier events matched in the subscription. Consider the following subscription.

Example 8 *Forward the quotes for any stock that increases monotonically for at least 30 minutes, and then does not return to its initial value for another hour afterwards.*

At first glance, this is a concatenation of two iterations, one that detects the initial monotonic run upwards, and the other which ensures that the stock never drops below its initial value. However, we cannot express this as two sequenced unary iterations of the form

$$(S_1)^{\star}_{\sigma_{\theta_1},\phi_1} \underset{\phi_2}{;} (S_2)^{\star}_{\sigma_{\theta_2},\phi_2}$$

The second iteration $(S_2)^{\star}_{\sigma_{\theta_2},\phi_2}$ will stop once the stock drops below its initial price. However the initial price is stored in the first iteration $(S_1)^{\star}_{\sigma_{\theta_1},\phi_1}$ and cannot be accessed by σ_{θ_2} as the iteration is unary. Thus we extend μ to a binary operation to allow the fullest collection of subscriptions possible.

3.2.4 Aggregates

Aggregates take sets of data and summarize them into a single value. In streaming data, we have two natural ways in which we can accumulate sets of data for aggregation. One is a set of data accumulated as a sequence of events over time, like our iteration operator. Another is a set of events that all happen at a single point in time.

Sequence Aggregation

We call the first type of aggregation *sequence aggregation* because it aggregates on a sequence events over time. Sequence aggregation fits naturally into our algebra, provided that we can implement it as a running aggregate while we iterate over the sequence of events. To implement such an aggregate, as in SQL, we need to create new attributes where the aggregate values are stored. More formally, an *attribute introduction function* g is a map that takes a set of attributes and produces a new set of attributes for aggregate values, computed from arithmetic combinations of existing attribute values and constants. For any event e, we let $g[\![e]\!]$ be e with those computed aggregate values added according to the rules of g. For example, suppose

$$e \;=\; \langle \text{IBM}, 80, \text{IBM}, 82; 1, 3\rangle \in S_1 \underset{\theta}{;} S_2$$

where S_1 and S_2 refer to the stock stream, and θ is the formula $S_1.\texttt{Name} = S_2.\texttt{Name}$. We let g be the map $\frac{S_1.\texttt{Price}+S_2.\texttt{Price}}{2} \mapsto \text{AVG}$. Then,

$$g[\![e]\!] \;=\; \langle \text{IBM}, 80, \text{IBM}, 82, 14.5; 1, 3\rangle$$

for the new data schema $(S_1.\texttt{Name}, S_1.\texttt{Price}, S_2.\texttt{Name}, S_2.\texttt{Price}, \text{AVG})$. Should g refer to any attribute not in e, then the value of $g[e]$ will be NULL.

Given an expression $\mathcal{E}$ and introduction function g, the *attribute introduction operator* α_g is defined as

$$\alpha_g(\mathcal{E}) \;=\; \{\, g[\![e]\!] \mid e \in \mathcal{E} \,\}$$

Together with μ, this operator gives us sequence aggregation. Consider an expression of the form

$$\alpha_{g_3}\big(\mu_{\alpha_{g_2}\circ\mathfrak{F},\theta}(\alpha_{g_1}(\mathcal{E}_1), \mathcal{E}_2)\big)$$

In this expression, α_{g_1} functions as an initializer, introducing one or more aggregate attributes and initializing their values. The operator α_{g_2} is an accumulator, updating the

aggregate values. Finally, α_{g_3} is an optional finalizer, adding new aggregate attributes to the schema or updating existing aggregate values, based on the existing attribute values in the output schema of μ. Notice that the aggregate attributes manipulated by g_i's $i = 1, 2, 3$ are always preserved during the computation of μ – they are never renamed or projected out.

For example, suppose we want the average of IBM stock over the past 52 weeks, as in Example 5. If we let $S_1, S_2 = S$ be our stream of stock quotes, we can compute the 52-week average with the expression

$$\sigma_{\text{DUR}=52\text{ weeks}}\left(\mu_{\alpha_{g_2},\text{TRUE}}(\alpha_{g_1} \circ \sigma_{S_1.\texttt{Name}=\text{IBM}}(S_1), \sigma_{S_2.\texttt{Name}=\text{IBM}}(S_2))\right) \tag{3.3}$$

where the introduction functions are

$$g_1 = \texttt{Price} \mapsto \text{AVG}, 1 \mapsto \text{COUNT}$$

$$g_2 = \text{COUNT} + 1 \mapsto \text{COUNT}, \frac{\text{COUNT} \times \text{AVG} + S_2.\texttt{Price}}{\text{COUNT} + 1} \mapsto \text{AVG}$$

Notice that within this iterated computation, the introduction functions can only refer to values computed in previous iterations. The new values computed in this iteration will update old values atomically because of the definition of iteration.

As with other subscriptions, the aggregate values computed above can be used in more complex subscriptions. For example, we express Example 5 as

$$\sigma_{S_3.\texttt{Price}>\text{AVG}}(\mathcal{E}\,;\,\sigma_{S_3.\texttt{Name}=\text{IBM}}(S_3))$$

where $S_3 = S$ is our stream of stock quotes and $\mathcal{E}$ is the subscription from (3.3).

Instance Aggregation

The second type of aggregation is called *instance aggregation* because it aggregates over a set of events that happen at a single instant in time. For example, suppose we

want the maximum 52-week average among all tech stocks. While we can use sequence aggregation to find the 52 week average for each stock, the 52-week periods for each stock are all simultaneous. Therefore, we need instance aggregation to compute their maximum.

Instance aggregation is very similar to traditional database aggregation. An *aggregation function* F maps a set of tuples to a single tuple. The instance aggregation operator β_F takes all of the events that occur at a single instance, applies F to their data values to produce a tuple $\overline{a}$, and produces the event with that data tuple. Formally,

$$\beta_F(S) = \left\{ \left\langle F(S_{\text{END}}); \text{END}, \text{END} \right\rangle \,\middle|\, S_{\text{END}} = \{\, \overline{b} \mid \left\langle \overline{b}; s, \text{END} \right\rangle \in S \,\} \right\}$$

Note that sequence aggregation takes all of the events that occur at precisely the same point in time (i.e. have the same END value). This means that it is not always applicable to events that are only approximately simltaneous, such as the 52-week averages of several tech stocks. Technically, those averages are events that are simultaneous with the last stock quote used to compute the average. While these stock quotes will be very close together, they are unlikely to be simultaneous. To get around this problem, we usually need an "anchor event" to sequence with the averages in order to make them simultaneous. Consider the following subscription.

Example 9 *Once Google exceeds $400, forward the tech stock with the maximum 52-week average.*

Here, the Google quote acts as our anchor event. To express this subscription, we first have to compute the 52-week averages of our tech stocks. If we let S_1 and S_2 be the stream of tech stocks, the subscription computing this average is

$$\mathcal{E} = \sigma_{\text{DUR=52 weeks}} \left(\mu_{\alpha_{g_2}, S_1.\texttt{Name}=S_2.\texttt{Name}}(\alpha_{g_1}(S_1), S_2) \right)$$

where the introduction functions g_i are the same as in (3.3). Example 9 is then

$$\beta_F\big(\sigma_{S_3.\texttt{Price}>\$400}(\mathcal{E}^{;}_{S_3.\texttt{Name}=\text{Goog}}(S_3))\big)$$

where F takes a set of stock quotes and returns the quote with the maximum 52-week average.

3.2.5 Expressing the Sample Subscriptions

In the previous section, we have seen how to use the various operators to express all of the subscription in Section 3.1. For convenience, we summarize all of these algebra expressions in Table 3.2. For the two RSS subscriptions, we assume all the blogs the user subscribes to consist of $\texttt{site}_1, \cdots, \texttt{site}_n$, and **contains**$(T, P)$ is the substring match operator that tries to find substring pattern P in text T. ID is the identity operation that has no effect on the input; it exists for those cases in which we do not want to perform any intermediate processing during iteration.

3.3 Processing Expressions

Given the algebra's similarity to regular expressions, finite automata would appear to be a natural implementation choice. We extend standard finite automata [47] in two ways. First, attributes of events can have infinite domains, e.g., text attributes, and therefore the input alphabet of our automaton, which is the set of all possible events, can be infinite as well. To handle this case, we associate each automaton edge with a *predicate*, and for an incoming event, this edge is traversed iff its predicate is satisfied by this event. Second, to be able to generate customized notification and to handle parameterized predicates over infinite domains, we need to store in each automaton instance the attributes and

Table 3.2: Algebraic Expressions for Sample Subscriptions

Example	Expression
1	$\sigma_{\texttt{Name}=\text{IBM}\wedge\texttt{Price}>\$100}(S)$
2	$\sigma_{S_1.price\leq\$100\wedge S_2.price>\$100}(\sigma_{S_1.name=\text{IBM}}(S_1)\,;\,\sigma_{S_2.name=\text{IBM}}(S_2))$
3	$\sigma_{1.05*S_1.\texttt{Price}<=S_2.\texttt{Price}}(S_1\,;_{S_1.name=S_2.name}\,S_2)$
4	$\sigma_{\text{DUR}\geq 30\text{ min}}\left(\mu_{\sigma_{S_2.\texttt{Price}>S_2.\texttt{Price.last}},S_1.\texttt{Name}=S_2.\texttt{Name}}(S_1,S_2)\right)$
5	$\sigma_{S_3.\texttt{Price}>\text{AVG}}(\mathcal{E}\,;\,\sigma_{S_3.\texttt{Name}=\text{IBM}}(S_3))$ where $\mathcal{E}=\sigma_{\text{DUR}=52\text{ weeks}}\left(\mu_{\alpha_{g_2},\text{TRUE}}(\alpha_{g_1}\circ\sigma_{\texttt{Name}=\text{IBM}}(S_1),\sigma_{\texttt{Name}=\text{IBM}}(S_2))\right)$
6	$\sigma_{S_1.\texttt{website}=\texttt{apple.com}}(S_1)\,;_{\textbf{contains}(S_2.\texttt{description},S_1.\texttt{link})}\,\sigma_\theta(S_2))$ where $\theta=\bigvee_{i\leq n}S_2.\texttt{website}=\texttt{site}_i$
7	$\mu_{\text{ID},\textbf{contains}(S_2.\texttt{description},S_1.\texttt{link.last})}\big(\sigma_{\theta\wedge\textbf{contains}(S_1.\texttt{description},\text{`Apple rumor'})}$ $(S_1),\sigma_\theta(S_2)\big)$ where $\theta=\bigvee_{i\leq n}S_2.\texttt{website}=\texttt{site}_i$

values of those events that have contributed to the state transition of this instance. These attributes and values are called *bindings*. To avoid overwriting the bindings of earlier events with that of later events, we also need an attribute renaming function for each edge so that when an event makes an automaton instance traverse that edge, the bindings in that event are properly renamed before being stored in the instance.

Our automata are nondeterministic. Whenever an automaton is in a state where it can traverse more than one edge for an incoming event, it nondeterministically traverses all these edges. If it cannot traverse any edge, the corresponding branch "dies". This is equivalent to having multiple *active instances* of the automaton explore the different branches, each branch corresponding to a prefix of the subscription sequence.

In this section, we will present a mechanism to translate algebra expressions into automata. Intuitively, for a given algebra expression, we first construct a parse tree, and then translate each tree node corresponding to a binary operator into an automaton node. In our mechanism any left-deep parse tree can be translated into a single automaton, referred to as a *left-deep* automaton. We first decribe how to handle left-deep expressions, and then generalize to arbitrary expressions at the end of this section (Section 3.3.4).

3.3.1 Automaton Example

First we present an example to illustrate a left-deep automaton.

Example 10 *Forward the quotes for any stock s, for which there is a monotonic decrease in price for at least 10 minutes, starting at a large trade ($\texttt{Vol} > 10,000$). The next quote of the same stock after this monotonic sequence should have a price 5% above the previously seen (bottom) price.*

After the first large trade of a stock, the automaton will begin looking for a monotonically decreasing sequence, then for a sudden increase in price. At any given moment, there might be several event sequences that satisfy some prefix of the subscription pattern.

For presentation of this example, we use a slightly simplified version of our actual automata. Our goal is to provide an intuitive understanding of the approach, before introducing the formal definition. We let S be the input stream of stock quotes, and assume for the purpose of this example that no two quotes in the stream have the same timestamp. To make the schemas of the intermediate results clear, we will use explicit renaming operators ρ_f to specify the schema at each step. Specifically, we define the

schema $\mathbb{X} = \{\texttt{Name}, \texttt{Price}\}$, $\mathbb{Y} = \{\texttt{company}, \texttt{minP.first}, \texttt{minP}, \texttt{finalP}\}$, and the renaming operators

$$\begin{aligned} f_1 &= (\texttt{Name}, \texttt{Price}) \mapsto (\texttt{company}, \texttt{minP}) \\ f_2 &= (\texttt{Price}) \mapsto (\texttt{finalP}) \\ f_3 &= (\texttt{minP.first}) \mapsto (\texttt{maxP}) \end{aligned}$$

The algebra expression for Example 10 is then

$$\rho_{f_3} \circ \pi_{\mathbb{Y}} \circ \sigma_{\theta_5} \Big(\sigma_{\theta_4} \big(\mu_{\sigma_{\theta_3}, \theta_2} \big(\rho_{f_1} \circ \pi_{\mathbb{X}} \circ \sigma_{\theta_1}(S), \rho_{f_1} \circ \pi_{\mathbb{X}}(S) \big) \big) \underset{\theta_2}{;} \rho_{f_2} \circ \pi_{\mathbb{X}}(S) \Big) \tag{3.4}$$

where the selection formula are

$$\theta_1 \equiv \texttt{Vol} > 10,000 \qquad \theta_2 \equiv \texttt{company} = \texttt{company.first}$$

$$\theta_3 \equiv \texttt{minP} < \texttt{minP.last} \qquad \theta_4 \equiv \text{DUR} \geq 10\,\text{min}$$

$$\theta_5 \equiv \texttt{finalP} > 1.05 * \texttt{minP}$$

The algebra expression is interpreted as follows. S_1 is obtained from S by selecting only large volume trades (θ_1), then projecting out the volume attribute and changing the attribute names (f_1). Hence S_1 contains only large trades and has data schema $(\texttt{company}, \texttt{minP})$. The μ operator searches for a monotonically decreasing sequence (θ_3) for the same stock, ignoring quotes from other companies (θ_2). During each iteration μ compares the current lowest price to the price of the incoming event. If a new minimum price is found, the concatenation overwrites the previously lowest price by the new one, otherwise the monotonic sequence has ended. The μ operator produces output events as soon as the duration constraint in θ_4 is satisfied. Finally, the $;_{\theta_2}$ operator finds the next quote for the same company (θ_2). If the price of that quote satisfies θ_5, the subscription produces an output event. As before, the renaming operator ρ_{f_2} ensures,

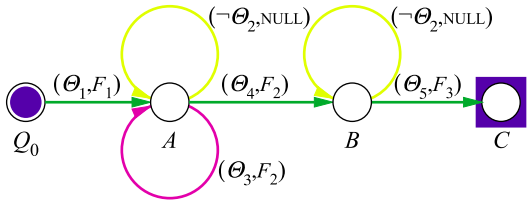


Figure 3.2: Automaton for Example 10

that the final price is added to the result as attribute `finalP`. ρ_{f_3} and the projection operator applied immediately before it transforms the output event to retain only four attributes `company`, `maxP`, `minP` and `finalP`.

The corresponding automaton is shown in Figure 3.2. We associate each edge with an edge predicate Θ_i and an attribute mapping F_j. The attribute mappings F_j are derived from the renaming functions f_j, except that they specify the entire schema, and not just the renamed attributes. They are defined as follows

$$F_1 = \texttt{Name} \mapsto \texttt{company}, \texttt{Price} \mapsto \texttt{minP},$$
$$\texttt{Name} \mapsto \texttt{company.first}, \texttt{Price} \mapsto \texttt{minP.first}$$
$$F_2 = \texttt{Name} \mapsto \texttt{company}, \texttt{Price} \mapsto \texttt{minP},$$
$$\texttt{company.first} \mapsto \texttt{company.first}, \texttt{minP.first} \mapsto \texttt{minP.first}$$
$$F_3 = \texttt{company} \mapsto \texttt{company}, \texttt{minP.first} \mapsto \texttt{maxP}, \texttt{minP} \mapsto \texttt{minP},$$
$$\texttt{Price} \mapsto \texttt{finalP}$$

Note that these mappings are not true functions; F_1 maps `Name` to both `company` and `company.first`. This is to support the value duplication necessary for initializing an iteration loop, as described in Section 3.2.3. Given these attribute mappings, we define

the edge predicates as follows.

$$\Theta_1 \equiv \texttt{Vol} > 10,000 \qquad \Theta_2 \equiv \texttt{Name} = \texttt{company}$$

$$\Theta_3 \equiv \Theta_2 \wedge \texttt{Price} < \texttt{minP} \qquad \Theta_4 \equiv \Theta_3 \wedge S.t_1 - A.t_0 \geq 10\,\text{min}$$

$$\Theta_5 \equiv \Theta_2 \wedge \texttt{Price} > 1.05 * \texttt{minP}$$

Again, these predicates are much the same as the selection predicates of (3.4), except that the attribute names are renamed after selection, not before, and we have conjuncted predicates together as appropriate. By $S.t_1$, we mean the end time of the end time of the event traversing the edge with this predicate; $A.t_0$ is the start time of the event stored at state A.

For this example, we suppose that we have the event stream illustrated in Table 3.3. Table 3.4 illustrates how the automaton processes these events. For an incoming event, the state of the automaton after processing it is indicated by the active automaton instances in the same row. The table headers show the data schema of the instances at a given automaton state. For readability, the timestamp attributes are not shown in the schema.

Table 3.3: Example Event Sequence

Event	(Name, Price, Vol)	Event	(Name, Price, Vol)
e_1	$\langle$IBM, 90, 15000; 9:10, 9:10$\rangle$	e_2	$\langle$IBM, 85, 7000; 9:15, 9:15$\rangle$
e_3	$\langle$Dell, 40, 11000; 9:17, 9:17$\rangle$	e_4	$\langle$IBM, 81, 8000; 9:21, 9:21$\rangle$
e_5	$\langle$MSFT, 25, 6000; 9:23, 9:23$\rangle$	e_6	$\langle$IBM, 91, 9000; 9:24, 9:24$\rangle$

Initially there is no active automaton instance, but the start state is always active by default. When e_1 arrives, the automaton checks if it satisfies Θ_1, the predicate on the edge emanating from the start state. This is the case, therefore it applies the attribute

Table 3.4: Example computation

Event	Instances at State A $\left(\texttt{company.first}, \texttt{minP.first}, \atop \texttt{company}, \texttt{minP}\right)$	Instances at state B $\left(\texttt{company.first}, \texttt{minP.first}, \atop \texttt{company}, \texttt{minP}\right)$	Instances at state C $(\texttt{company}, \texttt{maxP}, \texttt{minP}, \texttt{finalP})$
e_1	I_1 = ⟨IBM,90,IBM,90;9:10,9:10⟩		
e_2	I_1 = ⟨IBM,90,IBM,85;9:10,9:15⟩		
e_3	I_1 = ⟨IBM,90,IBM,85;9:10,9:15⟩ I_2=⟨Dell,40,IBM,40;9:17,9:17⟩		
e_4	I_1 = ⟨IBM,90,IBM,81;9:10,9:21⟩ I_2=⟨Dell,40,IBM,40;9:17,9:17⟩	I_3 = ⟨IBM,90,IBM,81;9:10,9:21⟩	
e_5	I_1 = ⟨IBM,90,IBM,81;9:10,9:21⟩ I_2=⟨Dell,40,IBM,40;9:17,9:17⟩	I_3 = ⟨IBM,90,IBM,81;9:10,9:21⟩	
e_6	I_2=⟨Dell,40,IBM,40;9:17,9:17⟩		I_3=⟨IBM,90,81,91;9:10,9:24⟩

mapping function g_1 to the attributes of e_1 and creates the resulting instance to state A.

The next event e_2 does not satisfy Θ_1, hence the start state does not create a new instance. For instance I_1 at state A, to determine if I_1 can traverse any outgoing edge, predicates Θ_2, Θ_3 and Θ_4 on the outgoing edges of A are evaluated with respect to e_2. This event only satisfies Θ_3 on the the rebind edge (self-loop below A). Therefore, the event traverses this edge and instance I_1 is updated by the mapping g_2. The result is shown in Table 3.4.

Event e_3 matches Θ_1; therefore a new instance I_2 is created at state A. For I_1, the concatenation of I_1 and e_3 only satisfies the predicate of the filter edge (top loop

of state A), because `company` $=$ IBM and `Name` $=$ Dell. Filter edges have special semantics—traversing them never updates the bindings of an instance. This is indicated in Figure 3.2 by the NULL value for the attribute mapping function.

The arrival of e_4 illustrates the non-determinism of the μ operator. e_4 is filtered for I_2 (the Dell pattern). However, for I_1 both Θ_3 and Θ_4 are satisfied (the duration condition in Θ_4 is now true). Hence I_1 non-deterministically traverses both the forward edge from A to B and the rebind edge of state A. This is implemented by cloning I_1 so that there is an instance to traverse each satisfied edge. In the example, clone I_3 under state B is created by applying g_2 to I_1 and the current event e_4.

Events e_5 and e_6 are processed similarly. For e_5, each of the instances traverses the corresponding filter edge. The interesting aspect of e_6 is its effect on instance I_1. I_1 concatenated with e_6 does not satisfy the predicate on forward or rebind edge of state A, therefore the instance is deleted. Notice how the nondeterminism ensures correct discovery of the IBM pattern for instance I_3 (events e_1, e_4, e_6 match it), but prevents any later arriving IBM event from generating another matching pattern starting with e_1, because I_1 has failed.

Overlooked Subtleties

While the previous example gives an excellent high level understanding of our automata, there are several subtleties that we have overlooked. First of all, as we saw in Section 3.2.4, the output of a subscription may contain several events with simultaneous end times. As Cayuga supports resubscription to the output of other queries, it must be designed to handle simultaneous events. However, simultaneous events can pose several difficulties. Consider our example above, but let there be another event

$e'_6 = \langle \text{IBM}, 80, 8000; 9{:}24, 9{:}24 \rangle$ at the same time as e_6. Even though this event fails to satisfy Θ_5, according to algebra semantics the automaton should still produce the output result with e_6. This suggests that forward edges (and rebind edges as well) are traversed so long as there *exists* a satisfying event. On the other hand, the same is not true for filter edges. According to the algebra semantics, the arrival of an additional event $e'_3 = \langle \text{IBM}, 99, 8000; 9{:}17, 9{:}17 \rangle$ at the same time as e_3 would cause I_1 to be deleted. Hence a filter edge should only be traversed if *all* simultaneously arriving events satisfy the filter predicate.

Second, there is a subtle issue in the implementation of duration constraints. Consider the simple subscription $\sigma_{\theta_1}(S; \sigma_{\theta_2}(S))$, where θ_1 and θ_2 are both predicates on duration. In this subscription, θ_2 refers to the duration of input events, while θ_1 refers to the duration of the composite events generated by the sequencing operator. In the automaton in Figure 3.2, we evaluate duration of the composite event, not the input event. Our final automaton must be able to distinguish between these two types of duration constraints and support them both.

Both of these issues are addressed in our formal automaton model.

3.3.2 The Formal Automata Model and Linear-Plus Expressions

The example in Section 3.3.1 demonstrates that Cayuga algebra expressions enforces a certain regularity on our automata structures. So long as our expressions are "left-deep" (i.e. all operators associate to the left), we do not need an arbitrary NFA; we only need an NFA that is acyclic (indeed, often linear) except for a few self-loops. Therefore, rather than directly translating arbitrary algebra expressions into automata, we will start with a simpler, but still powerful subset, which we refer to as *linear-plus expressions*. The

name "linear-plus" is inspired by the linear structure of the corresponding automata. In Section 3.3.4, we will show how to generalize our approach to handle arbitrary algebra expressions.

Definition 1 *We define the class of* linear-plus expressions $\mathcal{L}$ *as follows.*

- *For any base stream* S, $S \in \mathcal{L}$.
- *If* $\mathcal{E} \in \mathcal{L}$ *and* $\mathfrak{F}$ *is a unary expression formed from selection, projection, renaming, and aggregation, then* $\mathfrak{F}(\mathcal{E}) \in \mathcal{L}$.
- *If* $\mathcal{E}_1 \in \mathcal{L}$ *and* $\mathcal{E}_2 \in \mathcal{L}$, *then* $\mathcal{E}_1 \cup \mathcal{E}_2 \in \mathcal{L}$.
- *If* $\mathcal{E} \in \mathcal{L}$ *and* $\mathfrak{F}$ *is a unary expression, then* $\mathcal{E}$;$_\theta$ $\mathfrak{F}(S) \in \mathcal{L}$.
- *If* $\mathcal{E} \in \mathcal{L}$ *and* $\mathfrak{F}_1, \mathfrak{F}_2$ *are unary expressions, then* $\mu_{\mathfrak{F}_1,\theta}(\mathcal{E}, \mathfrak{F}_2(S)) \in \mathcal{L}$.

Before we formally define the automata, we give a brief overview of their structure. As we mentioned above, our automata will be acyclic, except for self-loops. Each node other than the start or end nodes will correspond to one of the binary operations ;$_\theta$ or $\mu_{\mathfrak{F},\theta}$. We encode these operations with three types of edges. *Forward* edges are those edges whose destination node is different from the source node, (e.g. the edge from A to B in Figure 3.2). Each node other than the end node has at least one such edge. Also, on each node other than the start node, there will be two self-loop edges, called the *filter* and *rebind* edge, respectively.

Filter edges correspond to the predicate θ in ;$_\theta$ and $\mu_{\mathfrak{F},\theta}$. In these expressions any event not satisfying θ (or satisfying $\neg\theta$) is supposed be filtered out. We remove these events with the filter edge, which is unique among the three types of edges in that the traversal of a filter edge does not modify the bindings of the instance. In our illustrations, we draw the filter edge on top of the node (see node A in Fig. 3.2). The predicate on a

filter edge (or *filter predicate*) corresponds to the negation of the selection predicate θ. This is illustrated in Figure 3.2 where Θ_2 is translated from θ_2 to account for the schema change. To ensure that events to be filtered do traverse the filter edge, the negation of the filter predicate (e.g. θ) appear in the forward and rebind edges of the same node as part of conjunction with the other predicates. This is illustrated in Figure 3.2 by the predicates Θ_3 and Θ_4 on node A. If a node corresponds to either the operation $;_{\text{TRUE}}$ and $\mu_{\mathfrak{F},\text{TRUE}}$, then no events are filtered. In this case, the filter predicate is FALSE, and we omit drawing the edge.

The rebind edge is draw below the node, as seen in node A of Figure 3.2. A rebind predicate corresponds to the selection formula component of $\mathfrak{F}$ in the operator $\mu_{\mathfrak{F},\theta}$. If a node corresponds to $;_\theta$ (sequencing), and not $\mu_{\mathfrak{F},\theta}$, then rebind predicate is FALSE. As with the filter edge, we omit drawing the rebind edge in this case. The selection of predicate for a rebind edge is illustrated in node A of Figure 3.2, which corresponds to the operations $\mu_{\sigma_{\theta_3},\theta_2}$. The rebind edge of node B has been omitted, as it corresponds to the operator $;_{\theta_2}$.

Formally, a *Cayuga automaton* is a directed multigraph $\mathcal{A}$ with the following properties:

- Nodes[2] are are marked as start, end, or intermediate. There is only one end state in each automaton.
- Every edge is marked as having either $\exists$-type or $\forall$-type.
- $\exists$-type edges are forward or rebind edges with label $(\exists, S, \Theta, g, F)$, where
  - Θ is any selection formula referencing attributes in either the input stream S, denoted as $S.\text{ATT}$, or the instances at the edge's source node q, denoted

[2]Nodes and states are used interchangeably.

as $q.\text{ATT}$. When the source of the attribute ATT is clear from the context, the prefix ($S.$ or $q.$) can be omitted.

  - S is the ID of the input stream of this edge.
  - g is an attribute introduction function that computes aggregate values.
  - F is an attribute mapping function that maps the attributes in S and the edge's source node to another set of attributes associated with the instances at the edge's destination node. Here F plays the role of both renaming and projection operators in the algebra. That is, if a certain attribute from the input stream or the source node is to be projected out, F will not map it to an attribute associated with the destination node.

- $\forall$-type edges are filter edges with label $(\forall, S, \Theta)$, where S and Θ are as for the $\exists$-type edges. $\forall$-type edges generate no output, and have no associated attribute introduction or mapping function.

To relate linear-plus expressions and Cayuga automata, we need to understand what it means to be the "output of a Cayuga automaton." Automata generally process finite strings, not unbounded streams. So formally, our automata will process *intervals* in a data stream. For any two time units $s_0 \leq s_1$, the interval $[s_0, s_1]$ in S is the set of all events $e \in S$ with $s_0 \leq e.\text{END} \leq s_1$. This is a finite set of events under a partial order. However, as events can be simultaneous, an interval is not necessarily linearly ordered like a string. However, this is not a problem, as events that are not ordered must be simultaneous, and $\forall$ and $\exists$ edges are designed to handle simultaneous events.

Each state of a Cayuga automaton has a fixed schema. An *instance* of a Cayuga automaton $\mathcal{A}$ is an event e associated with a state q, denoted as (e, q). Here q represents the current state of the automaton instance, and e represents the attribute bindings stored

in the instance buffer. At each time unit s corresponding to some event, the automaton reads all of events e' in the interval with $e'.\text{END} = s$, and determines whether or not they traverse an edge. An event traverses an $\exists$ edge if the tuple constructed by concatenating the current instance with the event satisfies the given edge predicate. An event traverses a $\forall$ edge if the same tuple does *not* satisfy the edge predicate.

If all of these events traverse the $\forall$ filter edge, then the instance remains unchanged. Otherwise, for each event e' being processed at time s, and each $\exists$ edge that it can traverse, we create a new instance of the automaton. To get this instance, we define the new bindings by the following steps.

- Add e to the bindings for the automaton, this making it the concatenation of the old instance and e. In those cases where e shares an attribute with the current instance, we overwrite the values of the instance with those from e.
- Apply g, the attribute introduction funtcion, to this result to get even more bindings.
- Apply F, the attribute mapping function, to project out values and get the schema of the final result.

In addition, this instance changes state according to the edge that was traversed. If no event at time s can traverse any of the edges of the automaton, then that instance is deleted. The output of an automaton at time s is the set of instances associated with its final state at time s.

With this definition of output, we can now relate automata and linear-plus expressions.

Theorem 1 *Let $\mathcal{E}$ be any linear-plus expression. There is a Cayuga automaton that computes $\mathcal{E}$. Furthermore, the number of states of this automaton is linear in the size of $\mathcal{E}$.*

3.3.3 Proof of Theorem 1

Our proof proceeds by induction on the definition of a linear-plus expression. Throughout this proof, we will make use of the following lemma, which gives us a convenient order in which to perform the unary operators.

Lemma 2 *Let $\mathfrak{F}$ be any unary expression formed from selection, projection, renaming, and aggregate, including multiple instances of each operator in any order. Then $\mathfrak{F}$ can be rewritten as $\mathfrak{F} = \pi_{\mathbb{X}} \circ \rho_f \circ \alpha_g \circ \sigma_\theta$.*

Proof 1 (Proof of Lemma 2) *The projection and renaming operators can be pulled to the left of a unary expression in the usual way. Furthermore, any two adjacent operators of the same type can be combined into a single operator by the appropriate operation on their subscripts. For example, two attribute maps can be joined by composing them, while two selections can be joined by combining their selection predicates with a conjunction. The only interesting case is $\sigma_\theta \circ \alpha_g$, where function g produces a new attribute y computed from existing attributes, and θ refers to y. This expression can be rewritten to $\alpha_g \circ \sigma_{\theta'}$, where θ' replaces the occurrence of each y in θ with the expression $g(y)$.*

Throughout our construction, we will often translate a selection predicate θ to an edge predicate Θ; this translation is to account for the slight differences between attribute names in our automata and in our algebra. Θ is similar to θ, except that for any

duration constraint DUR relop c in the algebra expression, the corresponding edge predicate will be either $S.t_1 - q.t_0$ relop c, or $S.t_1 - S.t_0$ relop c, where S is the input stream of the selection operator, and q is the source state of the edge where Θ is associated. We refer to these two translation options as I and II, respectively. In our construction below, we will describe which option to take in each case. Note that, when q is the start state, these two options are equivalent.

We translate $\pi_{\mathbb{X}} \circ \rho_f$ to an attribute mapping function F associated with an automaton edge as follows. Let the input stream of this unary expression have data schema $(a_1, \cdots, a_m)$. For each attribute $x \in \mathbb{X}$, if $x \notin \text{range}(f)$, F maps x to x; otherwise, F maps $f^{-1}(x)$ to x. Attributes not in $\mathbb{X}$ will not be mapped by F to anything.

For the base case of linear-plus expressions, consider the expression S. We implement this expression as a two-state automaton with a single edge labeled $(\exists, S, \text{TRUE}, \text{NULL}, \text{ID})$. This automaton is illustrated in Figure 3.3. Basically, each input event from S will create an output event of such an automaton with the same schema as the input, since the edge predicate is TRUE, and the attribute mapping function is identity function. The correctness of this automaton is trivial.

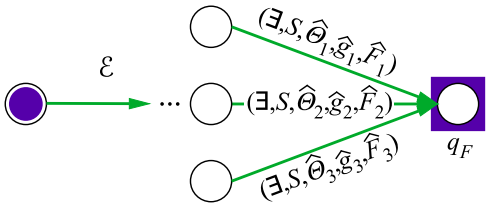


Figure 3.3: Automaton for S

Figure 3.4: Automaton for $\mathfrak{F}(\mathcal{E})$

Next we consider the case $\mathfrak{F}(\mathcal{E})$, where $\mathcal{E} \in \mathcal{L}$, and $\mathfrak{F} \equiv \pi_{\mathbb{X}} \circ \rho_f \circ \alpha_g \circ \sigma_\theta$ by Lemma 2. First we construct automaton $\mathcal{A}$ for $\mathcal{E}$. Take any forward edge leading to the end state, denoted as q_F, and suppose that edge has an existing label $(\exists, S, \Theta_i, g_i, F_i)$. We wish to replace this with a new edge $(\exists, S, \widehat{\Theta}_i, \widehat{g}_i, \widehat{F}_i)$, as shown in Figure 3.4. To get $\widehat{\Theta}_i$, we

compute the edge predicate Θ corresponding to the selection formula θ in $\mathfrak{F}$, via Option I. We want to conjunct this predicate with Θ_i to get a new predicate Φ_i. However, Θ may refer to attributes introduced by the function g_i, or renamed by F_i; in this automaton, the predicate Θ_i is evaluated before we apply either of these functions. Therefore, we must first construct the edge predicate Θ' by (1) renaming the attributes in Θ according to the inverse mapping F_i^{-1} and (2) replacing any attribute y introduced by g_i with the expression $g(\mathrm{y})$. $\widehat{\Theta}_i$ is then the conjunction of Θ' and Θ_i.

To get $\widehat{g}_i$, we must also rename the attributes in g according to the inverse mapping F_i^{-1}, to get a new introduction function g'. $\widehat{g}_i$ is then the composition of g_i and g', where g_i is applied first. Similarly, $\widehat{F}_i$ is the composition of F_i and F, where F is computed from $\pi_{\mathbb{X}} \circ \rho_f$ as above. Correctness of this automaton should be clear from our construction and the rules by which we update the bindings at each traversal.

The construction for $\mathcal{E}_1 \cup \mathcal{E}_2$, where $\mathcal{E}_1, \mathcal{E}_2 \in \mathcal{L}$, is given in Figure 3.5. Again we start with the automaton construction of $\mathcal{A}_i$ for each expression $\mathcal{E}_i$. We then merge the end states of $\mathcal{A}_1$ and $\mathcal{A}_2$. Correctness of this construction is the same as for regular expressions. While it is possible to also merge the start states of both automata, we do not want to do this in practice. In particular, we want to maintain the invariant that the set of out-going edges of any state reads the same input stream. This invariant is useful for efficient query processing, to be described in the next chapter.

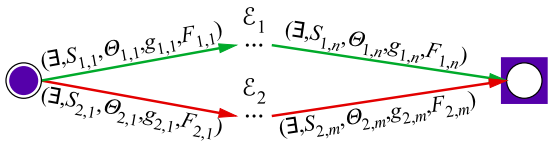


Figure 3.5: Automaton for $\mathcal{E}_1 \cup \mathcal{E}_2$

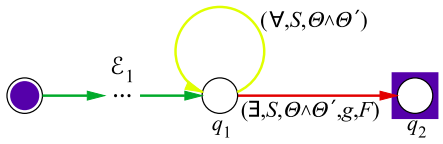


Figure 3.6: Automaton for $\mathcal{E}$;$_{\theta'}$ $\mathfrak{F}(S)$

Now we consider $\mathcal{E}$;$_{\theta'}$ $\mathfrak{F}(S)$, where $\mathcal{E} \in \mathcal{L}$, and $\mathfrak{F} \equiv \pi_{\mathbb{X}} \circ \rho_f \circ \alpha_g \circ \sigma_\theta$. We first build automaton $\mathcal{A}$ for $\mathcal{E}$ as shown in Figure 3.6. We denote the end state of $\mathcal{A}$ as q_1. As the

conditional sequencing operation $;_{\theta'}$ applies $\sigma_{\theta'}$ after $\mathfrak{F}$, we use Lemma 2 to rewrite

$$\sigma_{\theta'} \circ \mathfrak{F} \equiv \pi_{\mathbb{X}} \circ \rho_f \circ \alpha_g \circ \sigma_{\theta \wedge \theta''}$$

From θ'' we construct the edge predicate Θ'' via Option I. However, we will construct Θ from θ using Option II. Furthermore, we also need to rewrite θ into disjunctive normal form. For each conjunct in θ that refers to an attribute not in S, we replace that conjunct with FALSE. This ensures that the selection predicate will have the same behavior on nonexistent attributes as σ_θ does in Section 3.2.1.

Given Θ and Θ'', we add a new state q_2 to $\mathcal{A}$ and make it, not q_1, the final state. On q_1, the old final state, we add a loop edge $(\forall, S, \Theta \wedge \Theta'')$; recall that this edge is a filter edge, and it will remove events from stream S that do not qualify as the successor events to $\mathcal{E}$. The forward edge from q_1 to q_2 is labeled $(\exists, S, \Theta \wedge \Theta'', g, F)$, where F is constructed from $\pi_{\mathbb{X}} \circ \rho_f$ as above. If necessary, we remove any attributes from F and g that do not exist in the S according to the rules of Section 3.2.4.

To see that the output of this automaton is exactly $\mathcal{E} ;_{\theta'} \mathfrak{F}(S)$, take any event e output by the automaton in Figure 3.6. Then there is some interval $[s_0, s_1]$ such that (e, q_2) is an instance at the end of this interval. By definition, there are events e_1, e_2 such that $e_2.\text{END} = s_1$, (e_1, q_1) is an instance for $[s_0, r]$ with $r < s_1$, and $e_1 ; e_2 \models \Theta \wedge \Theta''$. Furthermore, e is the event we get when we apply $\Theta \wedge \Theta''$, g, and F to e_1, e_2, according to our rules for automaton traversal. Let r be least such timestamp with instance (e_1, q_1). As r is least, this instance is not produced by traversing the loop edge of q_1. So, e_1 is output by $\mathcal{A}$ on $[s_0, r]$, and thus $e_1 \in \mathcal{E}$ by induction.

The loop edge of q_1 does not add any new data values as it produces new instances; it only forwards the instance to the next stage. The only edge that can possibly add new data values is the forward edge from q_1 to q_2. Hence, $e_2 \in S$. We are given that $e_1 ; e_2 \models \Theta$. Our construction of Θ ensured that Θ only refers to attributes in e_2 (and it

was translated via Option II), so $e_2 \models \Theta$. Thus $e_2 \in \sigma_\theta(S)$. Let e_3 be the corresponding event in $\pi_{\mathbb{X}} \circ \rho_f \circ \alpha_g \circ \sigma_\theta(S)$ where we apply g and F to e_2. As g and F only refer to attributes in S, our construction guarantees that $e = e_1 ; e_2$. Thus $e \in \mathcal{E} ;_{\theta'} \mathfrak{F}(S)$, and so the output of our automaton is always in $\mathcal{E} ;_{\theta'} \mathfrak{F}(S)$. Running this argument in reverse gives the equivalence of $\mathcal{E} ;_{\theta'} \mathfrak{F}(S)$ and the automaton in Figure 3.6.

Finally, we present the construction for $\mu_{\mathfrak{F}_1,\theta'}(\mathcal{E}, \mathfrak{F}_2(S))$, which is illustrated in Figure 3.7, where $\mathfrak{F}_i \equiv \pi_{\mathbb{X}_i} \circ \rho_{f_i} \circ \alpha_{g_i} \circ \sigma_{\theta_i}$ for $i = 1, 2$. It is very similar to the construction for $\mathcal{E} ;_{\theta'} \mathfrak{F}(S)$. We first construct an automaton for $\mathcal{E}$ ending at state q_1. We know from Section 3.2.3 that the schema of $\mathfrak{F}_2(S)$ must be a subset of the schema of $\mathcal{E}$. Let the schema of q_1 be $\text{ATT}_1, \cdots, \text{ATT}_n$, and without loss of generality, the schema of $\mathfrak{F}_2(S)$ be $\text{ATT}_1, \cdots, \text{ATT}_k$, where $k \leq n$. We compose a new attribute mapping function to the forward edge leading to q_1 to change the schema of q_1 to $\text{ATT}_1.\texttt{first}, \cdots, \text{ATT}_n.\texttt{first}, \text{ATT}_1, \cdots, \text{ATT}_k$.

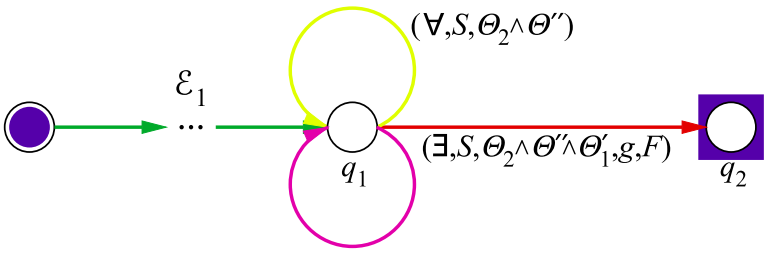


Figure 3.7: Automaton for $\mu_{\mathfrak{F}_1,\theta'}(\mathcal{E}, \mathfrak{F}_2(S))$

Next, we rewrite

$$\sigma_{\theta'} \circ \mathfrak{F}_2 \equiv \pi_{\mathbb{X}_2} \circ \rho_{f_2} \circ \alpha_{g_2} \circ \sigma_{\theta_2 \wedge \theta''}$$

according to Lemma 2. We put the filter edge $(\forall, S, \Theta_2 \wedge \Theta'')$ for filtering events from S that do not satisfy $\theta_2 \wedge \theta''$; the construction for Θ'' is modified so that attributes of the form $\text{ATT}.\texttt{last}$ are translated to $q_1.\text{ATT}$.

Again, we use Lemma 2 to rewrite

$$\mathfrak{F}_1 \circ \sigma_{\theta'} \circ \mathfrak{F}_2 \equiv \pi_{\mathbb{X}'} \circ \rho_{f'} \circ \alpha_{g'} \circ \sigma_{\theta_2 \wedge \theta'' \wedge \theta_1'}$$

We construct Θ_1' from θ_1' via Option I, renaming attributes of the form ATT.`last` as before. Furthermore, we construct F' from $\pi_{\mathbb{X}'} \circ \rho_{f'}$ as above; we do not need to account for the projection and renaming in (3.1) of Section 3.2.3, as this is handled by the way we replace values for attributes of the same name when we traverse an edge. Then we attach a rebind edge to q_1 with the label $(\exists, S, \Theta_2 \wedge \Theta'' \wedge \Theta_1', g', F')$. The forward edge leading to the new final state q_2 is identical to the rebind edge.

It is easy to see from the proof of Lemma 2 that $F' = F_1 \circ F_2$. Furthermore, g' is the union of g_1 and g_2, where any attribute in y in $g_1(\mathrm{x})$ from the domain of g_2 is replaced by $g_2(\mathrm{y})$. Thus, so long as we guarantee that Θ_2, g_2, and F_2 all only refer to attributes in S, the proof of correctness is similar to that for $\mathcal{E}\,;_{\theta'}\,\mathfrak{F}(S)$.

3.3.4 General Algebra Expressions

From the proof of Theorem 1, we see how to construct an automaton for any linear-plus expression. However, there are many algebra expressions, like $S_1\,;(S_2\,;\,S_3)$ that are not linear-plus. However, the output of any algebra expression is itself a stream, so we can express any algebra expression as a sequence of linear-plus expressions. For example, if we define $S_4 = S_2\,;\,S_3$, then $S_1\,;(S_2\,;\,S_3) = S_1\,;\,S_4$, where both $S_2\,;\,S_3$ and $S_1\,;\,S_4$ are linear-plus. In this case, we would call S_4 a *complex stream*, as it is a stream of complex events.

Naively, this suggests that we should be able to construct automata for general expressions by decomposing them into linear-plus expressions. We saw in the proof of

Theorem 1 that we can construct all our automata with exactly one start and one final state. This suggests that any directed edge can be replaced by any such automaton; we just identify the start and end of the automaton with the source and destination of the edge, respectively. For example, consider the automata in Figure 3.8 for $S_1 ; S_4$ and $S_2 ; S_3$. In this illustration, we identify the start and end states of the automaton for $S_2 ; S_3$ with states q_1 and q_2 in the automaton for $S_1 ; S_4$.

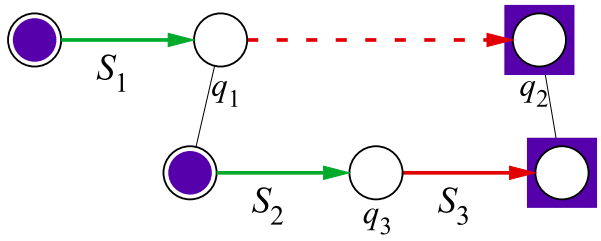


Figure 3.8: Attempt at Automaton for $S_1 ; (S_2 ; S_3)$

Unfortunately, the automaton in Figure 3.8 is not a correct automaton for the expression $S_1 ; (S_2 ; S_3)$. In fact, it is clear from the proof of Theorem 1 that the result is the automaton for $(S_1 ; S_2) ; S_3$ which is not, in general, equivalent to $S_1 ; (S_2 ; S_3)$ because sequencing is not associative. The problem is with how our automata handle simultaneous events. As we saw in Section 3.2.4, it is very possible to take a set of distinct events and make them simultaneous by sequencing them with a single anchor event. While all of these simultaneous events would traverse from state q_1 to q_2 in Figure 3.8, the base events are not simultaneous, and so only one of could traverse from q_1 to q_3.

For example, suppose S_1, S_2, and S_3 are the IBM, Microsoft, and Dell quotes, respectively, for the following stock stream.

$$S = \{\langle \text{IBM}, 80; 1, 1\rangle, \langle \text{MSFT}, 50; 2, 2\rangle, \langle \text{MSFT}, 49; 3, 3\rangle, \langle \text{Dell}, 24; 4, 4\rangle\}$$

This stream has been simplified for readability to contain only name and price, like the one in Section 3.2.3. The Dell quote is the immediate successor to both of the Microsoft

quotes, and therefore $S_2 ; S_3 = \{\langle \text{MSFT}, 50, \text{Dell}, 24; 2, 4\rangle , \langle \text{MSFT}, 49, \text{Dell}, 24; 3, 4\rangle\}$. However, the two events in $S_2 ; S_3$ are simultaneous, and so

$$S_1 ; (S_2 ; S_3) = \{\langle \text{IBM}, 80, \text{MSFT}, 50, \text{Dell}, 24; 1, 4\rangle, \langle \text{IBM}, 80, \text{MSFT}, 49, \text{Dell}, 24; 1, 4\rangle\}$$

However, there is only one Microsoft quote immediately succeeding the IBM quote, so

$$S_1 ; (S_2 ; S_3) = \{\langle \text{IBM}, 80, \text{MSFT}, 50, \text{Dell}, 24; 1, 4\rangle\}$$

This demonstrates that we cannot implement general algebra expressions in the naive way. We have two options. First, we could try to find a equivalent linear-plus expression for each general expression, and then construct the automaton for that. Failing this, could adapt the Cayuga engine to handle general expressions. In this section, we explore both these options.

Separation of General and Linear-Plus Expressions

Two algebra expressions are *semantically equivalent* if they give the same output for all choices of base stream(s). For example, if θ only references attributes in S_2, then $S_1 ; \sigma_\theta(S_2)$ and $S_1 ;_\theta S_2$ are semantically equivalent. Thus, if we could rewrite a general expression to a semantically equivalent linear-plus expression, we could construct an automaton for this expression. However, as the following theorem demonstrates, we cannot do this for all general expressions.

Theorem 3 ((Separation Theorem)) *There is no linear-plus query semantically equivalent to* $S_1 ; (S_2 ; S_3)$.

Proof 2 *While there are infinitely many linear-plus expressions to chose from, they are given by a simple recursive definition. We will construct a family of input data streams such that, for any linear-plus query* $\mathcal{E}$*,* $\mathcal{E}$ *and* $S_1 ; (S_2 ; S_3)$ *differ on at least one output.*

The family of input streams is constructed with two natural number parameters m and n as follows. Let the data schema of each of the streams S_i be $\emptyset$ (i.e. there are no data attributes; it is simply a stream of clock ticks). Let S_1 contain only one tuple of point timestamp (i.e. $e.\text{START} = e.\text{END}$) 0. Let S_2 contain tuples with point timestamps for all of the natural numbers. Finally, let S_3 contain two tuples of point timestamp $m+1$ and $m+n+2$. These streams are illustrated in Figure 3.9. We denote each set of streams as $\mathcal{S}(m,n)$.

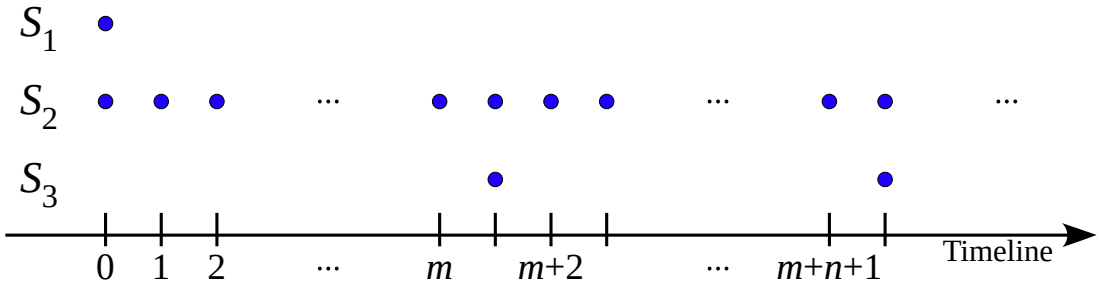


Figure 3.9: The Set of Streams $\mathcal{S}(m,n)$

The output of $S_1;(S_2;S_3)$ on $\mathcal{S}(m,n)$ consists of m tuples, where the i^{th} output is a concatenation of the S_1 tuple, the i^{th} tuple of the first m tuples in S_2, and the first tuple of S_3. No events in the output from $S_1;(S_2;S_3)$ will contain the second tuple of S_3.

Assume for a contradiction that there is a linear-plus query $\mathcal{E}$ semantically equivalent to $S_1;(S_2;S_3)$. We say that $\mathcal{E}$ is solid *if it is* not *the union of two other linear-plus expressions. We split the proof into two cases, depending on whether or not $\mathcal{E}$ is solid.*

***Case 1:** $\mathcal{E}$ is solid.*

Construct the automaton for the linear-plus expression. As $\mathcal{E}$ is solid, it is clear from the proof of Theorem 1 that the final state of this automaton must have only one incoming

edge, which we denote E. We let the source start of E be q_1 and the target state (the final state of the automaton) be q_2. This automaton is illustrated in Figure 3.10.

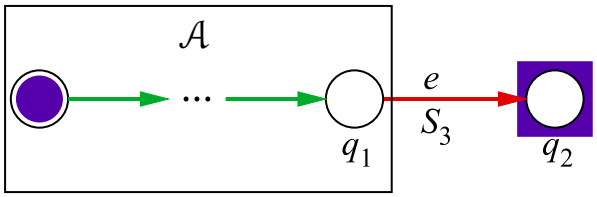


Figure 3.10: Automaton for Case 1

If this automaton is to produce the same output as S_1;(S_2; S_3), then timestamps of all its output must have the same end time as the first S_3 tuple in $\mathcal{S}(m,n)$. From the construction of our automaton, this can only happen if the event traversing the edge E to the final state has this end time, and so this edge must have input stream S_3. Furthermore, while there may be other edges in the automaton with input S_3, we can safely remove them without changing the output; every event reaching the final state must traverse edge E and so could not have traversed an earlier edge with S_3 as input.

This means that if we remove q_2 and E, and make q_1 the final state, we get an automaton with only inputs from S_1 and S_2. We call this automaton $\mathcal{A}$. As $\mathcal{E}$ produces the same output as S_1;(S_2; S_3), the instances associated with state q_1 should only involve the S_1 tuple and some of the first m tuples of S_2. None of the instances at q_1 should have any of the last n tuples of S_2, as these can be concatenated with the second S_3 tuple, producing different output.

For any m, n, let $I_{m,n,t}$ be the set of instances for state q_1 (i.e. the output of $\mathcal{A}$) at time t on the input $\mathcal{S}(m,n)$. For any m, n, $\mathcal{S}(m,n)$ has exactly the same values for S_1 and S_2. Hence $I_{m,n,t} = I_{m',n',t}$ for all t and $\mathcal{S}(m,n)$, $\mathcal{S}(m',n')$. Thus we let I_t represent the set of instances of $\mathcal{A}$ at time t on any of our inputs.

As we assumed that $\mathcal{E}$ correctly processes $S_1;(S_2;S_3)$, there are $m_0, k_0, n_0 \geq 1$, such that $\mathcal{E}$ and $S_1;(S_2;S_3)$ produce the same output on $\mathcal{S}(m_0+k_0+1, n_0)$. Consider the input $\mathcal{S}(m_0, k_0)$. At time m_0+k_0+1, the set of instances $I_{m_0+k_0+1}$ is the same on both inputs. Furthermore, both inputs have an S_3 event at time m_0+k_0+2. Thus it is clear from Figure 3.10 that the larger automaton $\mathcal{E}$ must produce an event with timestamp $[0, m_0+k_0+2]$ for both $\mathcal{S}(m_0+k_0+1, n_0)$ and $\mathcal{S}(m_0, k_0)$. However, this timestamp corresponds to the second event S_3 event in $\mathcal{S}(m_0, k_0)$. Thus $\mathcal{E}$ produces different output from $S_1;(S_2;S_3)$ on the input $\mathcal{S}(m_0, k_0)$, which is a contradiction.

Case 2: *Q' is not solid.*

We can regard the binary operator $\cup$ as polyadic, and thus let $\mathcal{E} \equiv \bigcup_{1 \leq i \leq k} \mathcal{E}_i$, where the $\mathcal{E}_i$ are all solid. Because of our construction in Theorem 1, any event output by the automaton for $\mathcal{E}$ must be output by the automaton for one of the $\mathcal{E}_i$.

Suppose again that $\mathcal{E}$ and $S_1;(S_2;S_3)$ agree on $\mathcal{S}(m_0+k_0, n_0)$ with $m_0, k_0, n_0 \geq 1$. Let $\mathcal{E}_i$ be one of the automata that outputs an event with timestamp $[0, m_0+k_0+2]$, and let I be the set of instances of this automaton at time m_0+k_0+1. Again, by the argument in Case 1, $\mathcal{E}_i$ has the same set of instances for $\mathcal{S}(m_0, k_0)$ at time m_0+k_0+1. So again $\mathcal{E}$ will produce an event with timestamp $[0, m_0+k_0+2]$ on $\mathcal{S}(m_0, k_0)$, which is a contradiction.

Implementing General Expressions

The Separation Theorem shows that no set of rewrite rules can be powerful enough to transform a general expression into an equivalent linear-plus one. This emphasizes the necessity for developing special query processing techniques for general expressions.

Instead of changing our automata model, we choose to implement general expressions through *resubscription*. In resubscription, automata are allowed to subscribe to the output of other automata as well as to base streams. To implement a general expression with resubscription, we break it up into a set of linear-plus expressions, and construct an automaton for each such expression. This is similar to the illustration in Figure 3.8 except that, instead of replacing the edge from q_1 to q_2, this edge takes the output from the other automaton as input.

We are able to treat other automata as data streams precisely because we went through the trouble ensuring that our automata could deal with simultaneous events of nontrivial duration. In fact, this functionality is the same as computing view streams, which is missing in event processing systems, but can be found in more powerful stream processing systems [11]. Indeed, the only issue to worry about is preventing circular references. We cannot have two automata $\mathcal{A}_1$ and $\mathcal{A}_2$ that subscribe to each other, as this may lead to non-termination. However, this cannot occur for resubscription on operator level. That is, if $\mathcal{A}_1$ and $\mathcal{A}_2$ are constructed from the same general expression, such case will not occur because the abstract syntax tree for an expression is acyclic. For resubscription on the automaton level, this can be prevented with a static check on the algebraic expressions to see whether they read the output streams of one another.

3.4 Remarks

In this chapter, we presented Cayuga, a novel solution for extended pub/sub applications. Cayuga extends previous work on event processing in several directions. First, it adds built-in support for parameterization, aggregates, selection over infinite domains, and support for arbitrary streams of events, inluding simultaneous events and events

with non-trivial duration. Second, a new automaton model for implementing algebra expressions efficiently is developed for Cayuga.

In the next chapter, we will discuss the challenges of implementing this automaton model, together with several MQO strategies. Also, we will present several initial performance results showing the efficacy of our approach.

CHAPTER 4

DESIGN AND IMPLEMENTATION OF THE CAYUGA SYSTEM

4.1 Introduction

As was described in the last chapter, a large class of both well-established and emerging applications can best be described as *event monitoring applications*. Event processing differs from general data stream management in two major aspects of the query workload. First, its has a distinct class of queries, which warrants special attention. In complex event processing, users are interested in finding matches to *event patterns*, which are usually sequences of correlated events. An important class of pattern is what we call a *safety condition*, where we want to ensure that between two events "nothing bad" happens. For example, between leaving the farm (start event) and arriving at the store (end event), fresh produce should not have spent more than 1 hour total above a temperature of 25 °C. Traditional data stream languages are not designed for event monitoring. While it is possible to express event patterns, this is cumbersome and results in queries that are almost impossible to read and to optimize.

Second, in complex event processing, there is usually a large number of concurrent queries registered in the event processing system. This is similar to the workload of publish/subscribe systems. In comparison, data stream management systems are usually less scalable in the number of queries, capable of supporting only a small number of concurrent queries.

In the last chapter, We have presented the event model, algebra, and automaton-based processing model of Cayuga, a general-purpose system for processing complex events on a large scale. Cayuga supports on-line detection of a large number of com-

plex patterns in event streams, and offers applications a unique combination of expressiveness and speed. Event patterns are expressed in a query algebra with well-defined formal semantics. This enables Cayuga to perform query-rewrite optimizations. All operators are composable, allowing Cayuga to build up complex patterns from simpler sub-patterns.

In this chapter, we describe the design and implementation of the Cayuga System. First, we present a query language designed for naturally expressing complex event patterns and illustrate its use through examples (Section 4.2). Second, after we review our automaton processing model and how queries are implemented by automata (Section 4.3), we describe the architecture of the Cayuga system (Section 4.4), and discuss important design decisions such as its garbage collection scheme. Next, we focus on the Cayuga query engine, and present novel Multi-Query Optimization techniques as well as how we leverage techniques from traditional pub/sub systems to achieve scalability in query processing (Section 4.5). The design decisions on the system architecture level, together with the MQO techniques in the query engine, enable Cayuga to gracefully handle many input streams, high event stream rates, and also a large number of continuous queries. In a thorough experimental study, we evaluate the scalability of our system both with the number of subscriptions and their complexity, we evaluate the efficacy of our MQO techniques, and we show the performance of our system with real data from our two example application domains (Section 4.6). We summarize related work in Section 5.6. Finally, Section 4.8 concludes our work on Cayuga.

4.2 The Cayuga Query Model

The Cayuga event model and query algebra have been described in the last chapter. Several unique features of the data model lead to important design decisions in the Cayuga system. In this section, we introduce the Cayuga Event Language (CEL), a query language based on the Cayuga query algebra.

4.2.1 Data Model

Like a traditional relational database system, Cayuga treats data as relational tuples, referred to as events. However, Cayuga is designed to monitor streams of events, not static tables. Thus, rather than sets of tuples, the Cayuga data model consists of temporally ordered *sequences* of tuples, referred to as event streams. Each event stream has a fixed relational schema. Each event in the stream has two timestamps, the start timestamp, denoted as t_0, and the end timestamp, denoted as t_1. Together they represent a duration interval, defined by $t_1 - t_0$. Events are serialized in order of t_1; for this reason, t_1 is also referred to as the "detection time" of a event. Because of the semantic issues discussed in White et al. [92], we consider events with the same detection time to be simultaneous, and guarantee to produce the same result regardless of the order of processing these simultaneous events. This guarantee is realized through *epoch-based processing* in Cayuga, as will be described in Section 4.4.3.

4.2.2 Query Language

The Cayuga Event Language is based on the Cayuga algebra [29], designed for expressing queries over event streams. It is a simple mapping of the algebra operators into a SQL-like syntax.

We introduce the query language through several examples. Our application domain for these examples is stock monitoring. We assume a stock ticker stream with the schema Stock(Name, Price, Volume). Each CEL query has the following simple form:

SELECT $< attributes >$
FROM $< stream\ expression >$
PUBLISH $< output\_stream >$

The SELECT clause in CEL is similar to the SQL SELECT clause. It specifies the attribute names in the output stream schema, as well as aggregate computation. Attributes can be renamed by AS constructs in the SELECT clause. The SELECT clause is optional; omitting it is equivalent to specifying SELECT *. The PUBLISH clause names the output stream, so other queries may refer to it as input. When this clause is omitted, the output stream is unnamed.

The simplest type of query is one that just reads all the events from one stream and forwards them to another, as shown in Example 11.

Example 11 *Suppose we want to select every input event from input stream Stock, and output it to a new stream named MyStock. We can formulate this query in CEL as follows.*

SELECT *

FROM *Stock*

PUBLISH *MyStock*

We refer to the expression in the FROM clause as a *stream expression*. This expression is the core of each query. A stream expression is composed using a unary construct, FILTER, and two binary constructs, NEXT and FOLD. Each of these constructs produces an output stream from one or two input streams. We introduce them by examples.

The FILTER{predExpr} construct selects those events from its input stream that satisfy the predicate defined by predExpr, as shown in the following example.

Example 12 *Suppose we want to select all IBM stock quotes whose price is above $83. We can formulate this query in CEL as follows.*

SELECT *Price* AS *IBMPrice*

FROM FILTER{*Name='IBM'* AND *Price* $>$ *83*}*(Stock)*

PUBLISH *IBMStock*

This query outputs only the Price attribute values of these events, not the stock Name or Volume. Furthermore, it renames the Price attribute to IBMPrice. The output stream is named IBMStock.

As in Cayuga algebra [29], a special attribute denoted as DUR, can be used in the predicate associated with FILTER. A predicate constraint on DUR is referred as a *duration predicate*. As is described in Section 4.2.1, attribute DUR of an event e takes the duration interval of e as the value. For example, an event e satisfies duration predicate DUR $>$ 10min if its duration interval is greater than 10 minutes.

CEL, like SQL, is compositional, allowing sub-queries in the FROM clause. For example, Example 13 gives an equivalent formulation of Example 12 using nested queries.

Example 13 *Again suppose we want to select all IBM stock quotes whose price is above $83. Another way to formulate this query in CEL as follows.*

SELECT *Price* AS *IBMPrice*
FROM FILTER{*Name='IBM'*}
*(*SELECT *
FROM FILTER{*Price* $>$ *83*}*(Stock))*
PUBLISH *IBMStock2*

The sub-query in the FROM *clause first produces all quotes whose Price is above $83. The top level query then further filters those quotes to retain only the quotes on IBM.*

Although not illustrated in Example 13, we can also publish the outputs of the nested sub-query as a separate stream of its own. We need only add an additional PUBLISH clause to the parenthesized sub-query. This enables one query formulation to produce multiple output streams.

Binary constructs in the stream expression allow us to correlate events over time. The first binary construct is NEXT{predExpr}. When applied to two input streams S_1 and S_2 as S_1 NEXT{predExpr} S_2, this construct combines each event e_1 from S_1 with the next event in S_2 which satisfies the predicate defined by predExpr and occurs after the detection time of e_1. When predExpr as the parameter of NEXT construct is omitted, it is by default set to TRUE. For example, S NEXT S returns a pair of consecutive events from stream S.

Example 14 *Suppose we want to match pairs of stock quotes, where the first is an IBM quote with a price above $83 and the second is the next Microsoft quote to appear in the stream. We can formulate this query in CEL as follows.*

SELECT *IBMPrice, Price* AS *MSFTPrice*
FROM *IBMStock*
NEXT{*Name='MSFT'*} *(Stock)*

Each event in the output stream of this query consists of a pair prices: an IBM price above $83 and the next MSFT price. In this query, the NEXT *construct reads two input streams, where the first stream IBMStock is produced by Example 12, whose schema contains only one attribute IBMPrice, and the second stream is Stock. Alternatively, we could have "inlined" the query from Example 12.*

A more powerful use of the NEXT construct exploits what we call "parameterization," the ability of the predExpr to refer to attributes from both its input streams, as in the following example.

Example 15 *Suppose we want to match pairs of stock quotes, where the first is an IBM quote with a price above $83 and the second is the next quote (of any stock) with a price above the IBM price from the first quote. We can formulate this query in CEL as follows.*

SELECT *IBMPrice, Price, Name*
FROM *IBMStock*
NEXT{$IBMPrice < Price$} *(Stock)*

Each event in the output stream of this query consists of an IBM price together with the name and price of the next stock to sell at a higher price.

In Examples 14 and 15 the two input streams of the NEXT construct have disjoint schemas. Of course this is not typical – the two input streams of a binary construct frequently contain identically named attributes. This situation happens to the binary join operator in relational algebra or SQL as well, for example in self-joins. When this happens to a binary construct, the reference to an attribute name in the predicate associated with the construct could become ambiguous, since the attribute could be from either one of the two input streams.

To address such reference ambiguity in CEL, we introduce special language constructs, referred to as *decorators*, to identify the streams from which attributes are taken. $1.foo refers to attribute foo in the first input stream of a binary construct, or the single input stream of a unary construct. Similarly, $2.foo refers to attribute foo in the second input stream of a binary construct. The decorator of an attribute can be omitted when it is in the schema of only one input stream. For simplicity of CEL, decorators are allowed only in predicate expressions, and cannot be used in the SELECT clause. It is helpful to view the SELECT clause as receiving one input stream whose schema is produced by the FROM clause.

The following example illustrates a simple use of decorators.

Example 16 *Suppose we want to match pairs of stock quotes with identical prices, and return the stock name of the second quote of each pair. We can formulate this query in CEL as follows.*

SELECT *Name*
FROM *(*SELECT *Price* FROM *Stock)*
NEXT{$*1.Price* = $*2.Price*} *(Stock)*

Each event in the output stream of this query consists the name of the second stock of a pair of quotes with identical prices. Note that the Name parameter occurs only in the schema of the second input to the NEXT *construct, so its use is unambiguous.*

While NEXT allows us to correlate two events, there are many situations where we need to iterate over an a-priori unknown number of events until a stopping condition is satisfied. This capability is supplied by the FOLD construct. A FOLD construct has the form FOLD{predExpr$_1$, predExpr$_2$, aggExpr}. The three parameters respectively denote (1) the condition for choosing input events in the next iteration; this plays the same role as predExpr in NEXT{predExpr} (2) the stopping condition for iteration, and (3) aggregate computation between iteration steps. Intuitively, FOLD is an iterated form of NEXT that looks for patterns comprising two or more events.

As with NEXT, we use decorators $1 and $2 respectively to refer to attributes in the first and the second input streams of FOLD. To refer to attributes in the *last* iteration of FOLD from the second stream, we use decorator $.

Example 17 *Suppose we want to find a monotonically increasing run of prices for a single company, where the run lasts for at least 10 stock quotes, and the first quote has a volume greater than 10000. We can formulate this query in CEL as follows.*

```
SELECT *
FROM FILTER{cnt > 10} (
  (SELECT *, 1 AS cnt FROM
  FILTER{Volume > 10000}(Stock))
    FOLD{$2.Name = $.Name, $2.Price > $.Price,
          $.cnt+1 AS cnt}
  Stock)
```

In Example 17 the first input stream of FOLD is produced by a FILTER construct that retains only quotes of volume greater than 10000. A new attribute cnt is added to that stream schema and initialized with value 1. The second input stream of FOLD contains all stock quotes. The first parameter of FOLD ensures that only stock quotes on the same company will be iterated over. The second parameter ensures that iteration will be stopped if the price of the current quote is not greater than that of the price in the last iteration. Finally, the third parameter computes the count aggregate, by adding 1 to the the value of cnt attribute in each iteration. Finally, for each output event of the FOLD construct, we run a filter over it to check whether its duration is greater than 1 hour.

To ensure valid iterations in FOLD construct, we maintain a *schema inclusion invariant* that the schema of its first input stream be a superset of the schema of its second input stream. Queries that violate this invariant will be rejected as being illegal. For more details on the formal semantics of FOLD and the invariant, we refer readers to [29].

Note that when the two input streams of a binary construct have identically named attributes, without proper renaming, the output stream of the binary construct will have duplicate attribute names, making the data semantics ambiguous. In relational algebra or SQL, this situation is addressed by explicitly renaming the output attribute names to make them distinct. Similarly for each NEXT construct, explict renaming can be used to avoid name collisions in its output schema. This is illustrated in Example 14, we had renamed the attribute Price in the first input stream schema of NEXT to IBMPrice.

For the FOLD construct, however, collisions cannot be avoided due to the schema inclusion invariant. For example, without attribute renaming, the output stream schema of Example 17 will contain two attributes named Price, among other duplicate attributes.

We use an automatic renaming scheme as follows to make sure the attributes in the output stream schema have distinct names. It applies to both NEXT and FOLD as follows.

For sub-expression R NEXT{predExpr} S, where R and S denote the input streams of the binary construct NEXT, if R and S do not have attribute name collision, the output schema will be a cross product of the two input schemas of R and S, and no renaming is performed. Otherwise, we rename *each* attribute a in R to a_1. For uniformity, even attribute names in R that do not appear in S are renamed this way. However, no renaming is performed on attribute names of S. It is possible that after this renaming operation, there are still duplicate attribute names in the output schema (consider the case when S has an attribute named a_1). In this case, the input query will be rejected as illegal.

For sub-expression R FOLD{unaryExpr, predExpr, aggExpr} S, for each attribute a that occurs in both R and S, the value of a in R will be stored in attribute a_1 in the output schema of the above sub-expression, and the value of attribute a in the latest iteration of S will be stored in attribute a in the output schema. Each attribute b in R but not in S will still be named b in the output schema. For example, the output schema of Example 17 is (Name_1, Price_1, Volume_1, cnt_1, Name, Price, Volume, cnt). Note that with attribute renaming, we avoided the use of hierarchical decorators such as $1.$1.foo to refer to attribute foo in the first stream S of expression (S NEXT S) NEXT S. Hierarchical decorators are therefore not allowed in CEL.

To illustrate the use of decorators and attribute renaming for NEXT construct, we give the following query formulation.

Example 18 *Suppose we want to match pairs of IBM stock quotes, where the first quote has a price above $83 and the second is the next IBM quote whose price is higher than*

the price in the first quote. We can formulate this query in CEL as follows.

SELECT *Price_1* AS *IBMPrice1, Price* AS *IBMPrice2*
FROM *(*FILTER{*Name='IBM'* AND *Price* $>$ *83*}*(Stock))*
NEXT{$*2.Price* $>$ $*1.Price*}
*(*FILTER{*Name='IBM'*}*(Stock))*

Note that attribute Price_1 in the SELECT *clause refers the attribute Price in the first input stream of* NEXT*, while the attribute Price comes from the second input stream.*

We believe our renaming scheme, when used appropriately, makes it easier to write queries by rendering explicit renaming unnecessary, and thus improves the user-friendliness of CEL.

4.3 Processing Model

In the last chapter, we showed that any left- associated Cayuga algebra expression can be implemented by a variant of a nondeterministic finite state automaton, referred to as a *Cayuga automaton*. Non-left-associated expressions can be broken up into a set of left-associated ones, and will therefore be implemented by a set of corresponding Cayuga automata. Since CEL is based on Cayuga Algebra, these results are applicable to CEL queries as well. In this section, we describe how to process CEL queries with Cayuga automata.

Cayuga automata generalize on traditional NFAs in two ways: (1) instead of a finite input alphabet they read arbitrary relational streams, with state transitions controlled

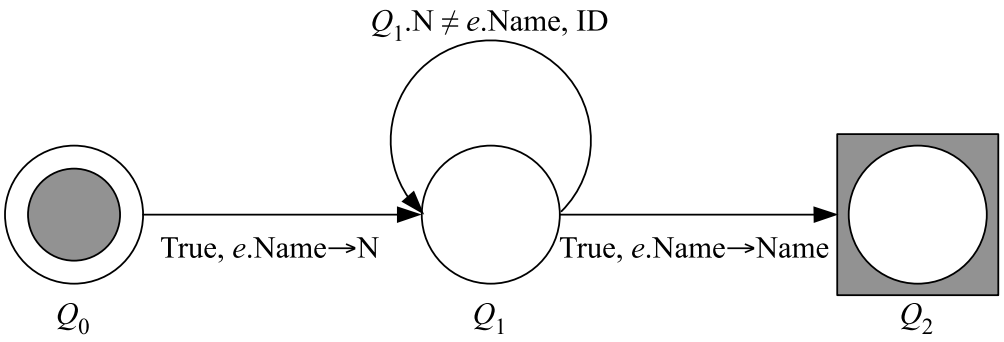


Figure 4.1: A Cayuga Automaton

using predicates; and (2) they can store data from the input stream, allowing selection predicates to compare incoming events to previously encountered events.

Each automaton state is assigned a fixed relational schema, as well as an input stream. All the out-going edges of a state read that input stream. Each edge, say between states P and Q, is labeled by a pair $\langle \theta, f \rangle$, where θ is a predicate over $\mathrm{schema}(P) \times \mathrm{schema}(S)$; and f, the *schema map*, is a partial function taking $\mathrm{schema}(P) \times \mathrm{schema}(S)$ into $\mathrm{schema}(Q)$. The Cayuga automata operate as follows. Suppose an automaton instance is in state P with stored data x (note x conforms to $\mathrm{schema}(P)$). Let an event e arrive on stream S such that $\theta(x, e)$ is satisfied. Then the machine nondeterministically transitions to state Q, and the stored data becomes $f(x, e)$.

In Cayuga automata, the self loop edges derived from predicates that filter events (i.e., predExpr in NEXT{predExpr}, and predExpr$_1$ in FOLD{predExpr$_1$, predExpr$_2$, aggExpr}) are called *filter edges*, and we call the associated predicates *filter predicates*; self loop edges derived from predicates that "rebind" attributees (i.e., predExpr$_2$ in FOLD{predExpr$_1$, predExpr$_2$, aggExpr}) are called *rebind edges*, and we call the associated predicates *rebind predicates*; other edges are *Forward* edges. We adopt the convention that filter edges are drawn on top of the states, and rebind edges below the states.

Intuitively, each intermediate state with a filter edge but no rebind edge implements a NEXT construct. Each intermediate state with both a filter and a rebind edge implements a FOLD construct. For example, an automaton implementing SELECT Name FROM (SELECT Name AS N FROM Stock) NEXT{$1.N = $2.Name} Stock is shown in Figure 4.1. According to the CEL formulation, The schema of state Q_1 has one attribute N, and the schema of Q_2 has one attribute Name. Both Q_0 and Q_1 read input stream Stock.

Predicates in CEL formulations are translated into automaton edge predicates in an obvious way. In particular, Attribute decorators in CEL are translated into prefixes $e.$ or $Q.$ for a given automaton edge predicate, depending on whether the attribute comes from the schema of the current event read by that edge, denote as e, or from the schema of the automaton state, denoted as Q, from which the edge emanates. For example, in Figure 4.1, the predicate on the forward edge between Q_1 and Q_2 compares the value of attribute N from state Q_1 with the value of attribute N from the input stream Stock.

Predicates in Cayuga automata are associated with edges. However, since there is always one filter edge for each state (except for start and end states), we can associate predicates on filter edges with automaton states without ambiguity. Similarly, since there is at most one rebind edge for each state, associating rebind edge predicates with automaton states is also not ambiguous.

Note that the predicate on a filter edge is the *negation* of the corresponding filter predicate in CEL formulation. For example, in Figure 4.1, the predicate on the filter edge associated with state Q_1 is Q_1.N $\neq e$.Name, while its corresponding filter predicate in the CEL formulation is $1.N = $2.Name. In the following text, to avoid ambiguity, we avoid using the term filter edge predicate. To be consistent with the notion of a filter predicate in CEL formulations, in the context of automaton edge predicates, we use the

term *filter predicate associated with state* Q to refer to the negation of the predicate of the filter edge associated with Q. For example, in Figure 4.1, the filter predicate associated with state Q_1 is $Q_1.\text{N} = e.\text{Name}$. Recall that we refer to the predicate on a rebind (resp. forward) edge as its rebind (resp. forward) predicate.

The schema maps of Cayuga automata are also constructed from the CEL formulations in an obvious way. Note that the schema map associated with a filter edge is always the identity function, and our implementation exploits this fact.

Any Cayuga automaton maintains the following invariants on its edge predicates.

- For any automaton instance I under state P, if the current event together with I satisfy the predicate of a forward edge from state P to Q, then they must together satisfy the filter predicate associated with state P.
- For any automaton instance I under state P, if the current event together with I satisfy the predicate of a forward edge from state P to Q, then they must together satisfy the rebind predicate associated with state P, if there is one.
- For any automaton instance I under state P, if the current event together with I satisfy the rebind predicate associated with state P, then they must together satisfy the filter predicate associated with state P.

A consequence of these invariants is that for any automaton instance I under state P, if the current event together with I does not satisfy the filter predicate associated with state P, then none of the predicates on the rebind or forward edges associated with state P will be satisfied. Therefore the instance I must traverse the filter edge of P and is unmodified (due to the identity schema map of the filter edge). In this case we say instance I is *not affected* by the current event. Otherwise, if the current event together with I satisfies the filter predicate of P, we say I is *affected* by the current event.

These invariants can be easily realized in the implementation by predicate conjunctions. For example, the first invariant could be realized by attaching the filter predicate of state P as a conjunct to the predicate of each forward edge leaving P, and to the rebind predicate associated with P, if there is one. With an understanding of these invariants, to simplify the presentation, in the automaton figures we usually do not duplicate filter predicates on forward or rebind predicates. For example, in the automaton shown in Figure 4.1, the predicate of the forward edge from Q_1 to Q_2 has semantics Q_1.N = e.Name. However, we decide not to show the filter predicate of Q_1 as a conjunct in this forward predicate, and therefore denote the forward predicate as TRUE.

Complete details of the NFA construction can be found in [29].

4.3.1 Automaton Example

To illustrate how Cayuga automata process a query, we present an extended example.

Example 19 *Suppose we want to record the quotes for any stock s, for which there is a monotonic decrease in price for at least 10 minutes, and which started at a large trade (Volume $>$ 10,000). Furthermore, suppose we are only interested in those monotonic runs that which are followed by one last stock quote whose prices is 5% above the previously seen (bottom) price. In other words, the stock has been steadily decreasing, but now shows signs of rising again. We can formulate this query in CEL as follows.*

SELECT *Name, MaxPrice, MinPrice, Price* AS *FinalPrice*
FROM
  FILTER{DUR$\geq$ *10min*} *(*
    *(*SELECT *Name, Price_1* AS *MaxPrice, Price* AS *MinPrice*

FROM FILTER{*Volume* > *10000*}*(Stock))*

FOLD{$*2.Name* = $*.Name,* $*2.Price* < $*.Price,* }

Stock)

NEXT{$*2.Name* = $*1.Name* AND

$*2.Price* >*1.05**$*1.MinPrice*}

Stock

In this formulation, the FOLD construct searches for a monotonically decreasing sequence for the same stock, ignoring quotes from other companies. During each iteration, the current lowest price is compared to the price of the incoming event. If a new minimum price is found, the concatenation overwrites the previously lowest price by the new one, otherwise the monotonic sequence has ended. When the duration constraint FILTER{DUR$\geq$ 10min} evaluated on the output of FOLD is satisfied, a complex event is output denoting a mononotically decreasing sequence that lasts for no less than 10 minutes. Finally, the NEXT construct finds the next quote for the same company. If the price of that quote is %5 above the previous price, the query produces an output event. The output event retains only four attributes Name, MaxPrice, MinPrice, and FinalPrice.

The corresponding automaton is shown in Figure 4.2. We associate each edge with an edge predicate θ_i and an attribute mapping F_j. They are defined as follows.

$$F_1 = e.Name \mapsto Name\_1, e.Price \mapsto Price\_1,$$
$$e.Name \mapsto Name\_n, e.Price \mapsto Price\_n$$
$$F_2 = ID : Schema(A) \mapsto Schema(A)$$
$$F_3 = A.Name\_1 \mapsto Name\_1, A.Price\_1 \mapsto Price\_1,$$
$$e.Name \mapsto Name\_n, e.Price \mapsto Price\_n$$
$$F_4 = e.Name \mapsto Name, A.Price\_1 \mapsto MaxPrice,$$
$$e.Price \mapsto MinPrice$$

$$F_5 = ID : Schema(B) \mapsto Schema(B)$$
$$F_6 = e.Name \mapsto Name, B.MaxPrice \mapsto MaxPrice,$$
$$B.MinPrice \mapsto MinPrice, e.Price \mapsto FinalPrice$$

Note that these mappings are not true functions; F_1 maps Name to both Name_1 and Name_n. This is to support the value duplication necessary for initializing an iteration loop. Given these attribute mappings, we define the edge predicates as follows.

$$\theta_1 \equiv e.Volume > 10,000$$
$$\theta_2 \equiv e.Name = A.Name\_n$$
$$\theta_3 \equiv e.Price < A.Price\_n$$
$$\theta_4 \equiv e.t_1 - A.t_0 \geq 10\,\text{min}$$
$$\theta_5 \equiv e.Name = B.Name$$
$$\theta_6 \equiv e.Price > 1.05 * B.MinPrice$$

The predicates on automaton edges are much the same as the predicates in CEL formulation, except that the attribute names are decorated with the sources where they

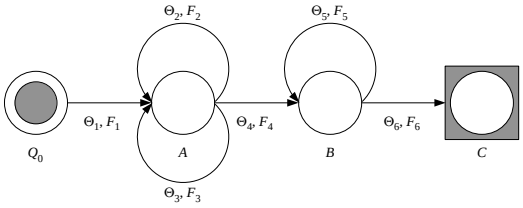


Figure 4.2: Automaton for Example 19

Event	(Name,Price,Volume)
e_1	$\langle$**IBM**, 90, 15000; 9:10, 9:10$\rangle$
e_2	$\langle$**IBM**, 85, 7000; 9:15, 9:15$\rangle$
e_3	$\langle$**Dell**, 40, 11000; 9:17, 9:17$\rangle$
e_4	$\langle$**IBM**, 81, 8000; 9:21, 9:21$\rangle$
e_5	$\langle$**MSFT**, 25, 6000; 9:23, 9:23$\rangle$
e_6	$\langle$**IBM**, 91, 9000; 9:24, 9:24$\rangle$

Figure 4.3: Example Event Sequence

come from. By $e.t_1$, we mean the end time of the event traversing the edge with this predicate; $A.t_0$ is the start time of the instance stored at state A.

After the first large trade of a stock, the automaton begins looking for a monotonically decreasing sequence, then for a sudden up-move in price. At any given moment in time, there might be several event sequences that satisfy some prefix of the subscription pattern.

For this example, we suppose that we have the event stream illustrated in Figure 4.3. Figure 4.4 illustrates how the automaton processes these events. For an incoming event, the state of the automaton after processing it is indicated by the active automaton instances in the same row. The table headers show the data schema of the instances at a given automaton state. For readability, the timestamp attributes are not shown in the schema.

Initially there is no active automaton instance, but the start state is always active by default. When e_1 arrives, the automaton checks if it satisfies θ_1, the predicate on the edge emanating from the start state. This is the case, therefore it applies the attribute mapping function F_1 to the attributes of e_1 and creates the resulting instance I_1 under state A.

Event	Instances at State A (Name_1,Price_1, Name_n, Price_n)	Instances at state B (Name,MaxPrice, MinPrice)	Instances at state C (Name, MaxPrice, MinPrice, FinalPrice)
e_1	$I_1 = \langle \text{IBM}, 90, \text{IBM}, 90; 9{:}10, 9{:}10\rangle$		
e_2	$I_1 = \langle \text{IBM}, 90, \text{IBM}, 85; 9{:}10, 9{:}15\rangle$		
e_3	$I_1 = \langle \text{IBM}, 90, \text{IBM}, 85; 9{:}10, 9{:}15\rangle$ $I_2 = \langle \text{Dell}, 40, \text{IBM}, 40; 9{:}17, 9{:}17\rangle$		
e_4	$I_1 = \langle \text{IBM}, 90, \text{IBM}, 81; 9{:}10, 9{:}21\rangle$ $I_2 = \langle \text{Dell}, 40, \text{IBM}, 40; 9{:}17, 9{:}17\rangle$	$I_3 = \langle \text{IBM}, 90, 81; 9{:}10, 9{:}21\rangle$	
e_5	$I_1 = \langle \text{IBM}, 90, \text{IBM}, 81; 9{:}10, 9{:}21\rangle$ $I_2 = \langle \text{Dell}, 40, \text{IBM}, 40; 9{:}17, 9{:}17\rangle$	$I_3 = \langle \text{IBM}, 90, 81; 9{:}10, 9{:}21\rangle$	
e_6	$I_2 = \langle \text{Dell}, 40, \text{IBM}, 40; 9{:}17, 9{:}17\rangle$		$I_3 = \langle \text{IBM}, 90, 81, 91; 9{:}10, 9{:}24\rangle$

Figure 4.4: Example computation

The next event e_2 does not satisfy θ_1, hence the start state does not create a new instance. For instance I_1 at state A, to determine if I_1 can traverse any outgoing edge, predicates θ_2, θ_3 and θ_4 on the outgoing edges of A are evaluated with respect to e_2. This event satisfies θ_3 on the rebind edge associated with state A. Therefore, the event traverses this edge and instance I_1 is updated by the mapping F_3[1]. The result is shown in Table 4.4.

Event e_3 matches θ_1; therefore a new instance I_2 is created at state A. For I_1, I_1 and e_3 together do not satisfy θ_2, the filter predicate associated with state A, because in I_1 the Name_n attribute has value IBM, while in e_3 the Name attribute has value Dell. I_1 is therefore not affected by this event.

[1]Technically, an automaton instance traversing the rebind edge creates a new instance, and then the original instance will be deleted after processing this event. We use the term "update" here to simplify the presentation.

The arrival of e_4 illustrates the non-determinism of the FOLD construct, implemented by state B. e_4 is filtered for I_2 (the Dell pattern). However, for I_1 both θ_3 and θ_4 are satisfied (the duration condition in θ_4 is now true). Hence I_1 non-deterministically traverses both the forward edge from A to B and the rebind edge of state A. In the example, we update to content of I_1 with the content of e_4, and create an new instance I_3 under state B by applying F_3 to I_1 and the current event e_4.

Events e_5 and e_6 are processed similarly. For e_5, each of the instances traverses the corresponding filter edge, and are thus not affected. The interesting aspect of e_6 is its effect on instance I_1. I_1 together with e_6 satisfies the filter predicate associated with state A, but does not satisfy the forward or rebind predicates of state A; therefore the instance is deleted. Notice how the nondeterminism ensures correct discovery of the IBM pattern for instance I_3 (events e_1, e_4, e_6 match it), but prevents any later arriving IBM event from generating another matching pattern starting with e_1, because I_1 has failed.

4.3.2 Additional Subtleties

While the previous example gives an good high level understanding of our automata, there are several subtleties that it fails to illustrate. First of all, there might be simultaneous events in a stream, produced either by an external stream source or by a Cayuga query. Simultaneous events can pose several difficulties in ensuring correctness of automaton processing. Consider our example above, but let there be another event $e'_6 = \langle \text{IBM}, 80, 8000; 9{:}24, 9{:}24 \rangle$ which has the same detection time as e_6, and is processed by the automaton between e_5 and e_6. Even though this event fails to satisfy θ_5, we cannot delete I_3 immediately, since according to the Cayuga query semantics, the

output event produced by I_3 together with e_6 should not be affected by the presence of e'_6. This suggests that after processing the current event e, we cannot delete those automaton instances that did not traverse their associated filter edges immediately, since if any following events are simultaneous with e, these automaton instances should remain visible while processing these events. We handle simultaneous events correctly by using epoch-based query processing, to be described in Section 4.4.3.

As stated earlier, not every Cayuga query can be implemented by a single automaton. In order to process arbitrary queries, Cayuga supports *resubscription*. Resubscription is similar to pipelining – the output stream from a query is used as the input stream to another query. Because of resubscription, query output must be produced in real time. Each tuple output by a query has the same detection time as the last input event that contributed to it, and thus its processing (by resubscription) must take place in the same epoch in which that event arrived. This decision motivates the Cayuga Priority Queue, described in Section 4.4.2 and the "pending instance lists" described in Section 4.4.3.

4.4 The Cayuga System

In this section, we describe the Cayuga system architecture, and we explain how we efficiently implement large number of automata.

4.4.1 Architecture

The Cayuga system architecture is shown in Figure 4.5. We first describe the control and data flow in Cayuga, and how components interact at a high level. We then focus on describing the major components of the system in more detail.

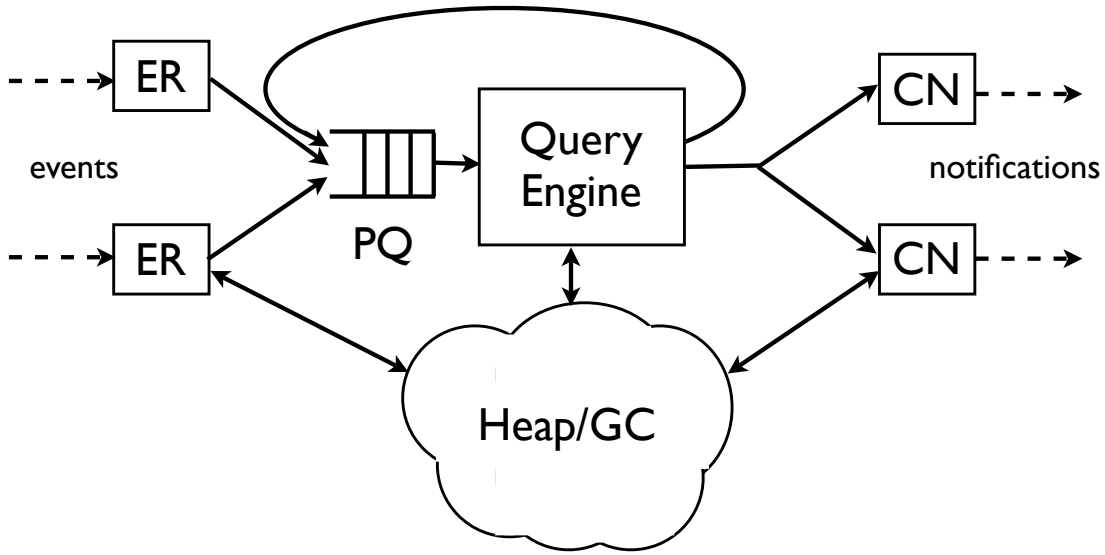


Figure 4.5: Cayuga System Architecture

External events arriving at the Cayuga system are received by Event Receivers (ERs), each of which runs in a separate thread, receiving events from a particular source. The ER threads are responsible for deserializing arriving events, assigning timestamps if necessary, internalizing them in the Cayuga Heap and inserting them on the input Priority Queue (PQ). The ERs and PQ are described in Section 4.4.2.

The Cayuga Query Engine is a single thread that is responsible for most of the query processing work. The engine dequeues events from the PQ in detection time order. It performs all indicated automaton state transitions, using several index structures to achieve high throughput. For each automaton instance reaching a final state, it enqueues a new event on the PQ if required for resubscription, and passes events to the appropriate Client Notifier threads (CNs). The engine is described in Section 4.4.3, and CNs are described in Section 4.4.4.

Cayuga uses a customized Memory Manager with a Garbage Collector (GC) to fa-

cilitate efficient object sharing for high performance and a small memory footprint. The Memory Manager is described in Section 4.4.5.

4.4.2 Event Receivers and Priority Queue

Cayuga has multiple Event Receiver threads. Each ER thread converts events arriving from an external data source (such as a TCP stream or a sensor device) into a sequence of internalized Cayuga events. Since the external data sources may encode data in different ways, there are multiple ER classes with deserialization methods specific to the data sources.

Internalized events have a common format, designed to allow sharing of complex data such as large string bodies or user-defined types. Shared data resides in the garbage-collected Cayuga Heap, discussed in Section 4.4.5. This design enables us to manipulate events and automaton instances using "shallow copy" operations and exploit efficient block move procedures whenever possible.

Newly-internalized events are inserted on the Priority Queue (PQ) which is ordered by event detection time. The events are later dequeued and processed by the Query Engine, described in Section 4.4.3.

As mentioned above, the Cayuga query model requires that events be processed in detection time order. Thus, the system must correct for clock skew between data sources, as well as for network delay and reordering. We address this problem as follows. Let T be an *a priori* bound on the out-of-orderedness of the input streams. That is, when an incoming event with timestamp t arrives at the Cayuga system, it is guaranteed that the system's local clock is at most $t + T$. With T so defined, proper ordering can be

achieved by buffering events in the PQ until they appear to be at least T time units old before allowing them to be dequeued. Specifically, a dequeue operation will block (or return *empty*) until the smallest detection time of any event in the queue is less than $c - T$, where c is the current Cayuga system clock value.

If an arriving event has a timestamp smaller than the timestamp of some event previously dequeued from the PQ, the event is ignored. Thus, coping with high variance in arrival times requires a large value for T. Note that increasing T adds latency but does not affect throughput.

Our use of a fixed global parameter T could be overly conservative in some cases. A more sophisticated (but less efficient) approach is described in [84].

Some data sources do not provide timestamps. For such data sources, the ER assigns to each arriving event a *point timestamp* – a 0-length interval – with detection time equal to the current Cayuga system clock. Clearly, a stream with such locally-generated timestamps can inter-operate with other streams whose timestamps are provided by the data sources.

Note that if *all* streams have locally-generated timestamps, then a suitable value for the global parameter T is 0. In this case, the PQ behaves as a FIFO queue except for resubscription processing as described in Section 4.4.3.

4.4.3 The Cayuga Query Engine

The Cayuga Query Engine processes internalized Cayuga events drawn from the Priority Queue, by executing state transitions of Cayuga automata and generating output events whenever final states are reached. Output events can be fed back into the PQ

for resubscription, or they can be passed to CN threads and forwarded to subscribers as described in Section 4.4.4. In the following subsections we overview the query engine. We then describe the details as well as a few important MQO techniques in Section 4.5.

Query Representation

The external representation of a Cayuga query uses an XML format we call Automaton Intermediate Representation (AIR). The AIR encoding of automaton states and edges is basically straightforward; however, the following features are worthy of note:

- Final states are explicitly identified for subscribing clients and for resubscription.
- Filter edges are explicitly identified, as they require different treatment from FR edges.
- Edge predicates and schema maps (as discussed in Section 4.3) are encoded as programs for a specialized bytecode interpreter discussed below.
- Predicate conjuncts that should be indexed are flagged appropriately (our indexing strategy is described in Section 4.4.3).

The AIR format is intentionally rather low-level, and can represent arbitrary collections of automaton states. This design decision enhances modularity – the Query Engine deals only with automaton execution, and is not concerned with the translation of algebra expressions into automata, or with multi-query optimizations involving automaton rewriting or sharing of sub-automata by resubscription. The only form of query optimization performed by the Engine is to merge manifestly equivalent states during AIR file loading.

When Cayuga loads an AIR file, the resulting internal data structure has an explicit representation of automaton states and edges. The Cayuga automaton model is nondeterministic. Thus, during query evaluation there may be many active instances of query automata, each in a different state and with different stored data. Our internal representation associates a list of *instance objects* with each automaton state, as depicted in Figure 4.6(a). An instance object associated with state Q represents an instance of the query automaton in state Q. The stored data of the automaton instance (conforming to the schema of Q) is contained in the instance object.

Edge predicates and schema maps are represented internally by bytecode interpreter programs, essentially copied from their AIR representation. The Cayuga bytecode interpreter is a straightforward stack machine specialized for efficient "short-circuit" evaluation of conjunctions of atomic formulas, and for efficient construction and copying of instance objects. The interpreter (as well as the AIR format) is designed to be extensible, so it can easily support new user-defined predicates (UDFs) and event schemas including new user-defined types (UDTs).

Simultaneity and Epochs

Cayuga's data model allows events with simultaneous detection times. To handle such events correctly, we use an *epoch-based* processing strategy. During epoch t, all events with detection time t are processed, but it is critical that no new automaton instance created during epoch t can be visible to other events detected in the same epoch. To achieve this effect, we put newly-created instances under a state into a separate *pending instance* list. At the end of each epoch, we perform *instance installation*, atomically merging each state's pending instance list with the state's surviving instances. This process is depicted in Figure 4.6(a-c).

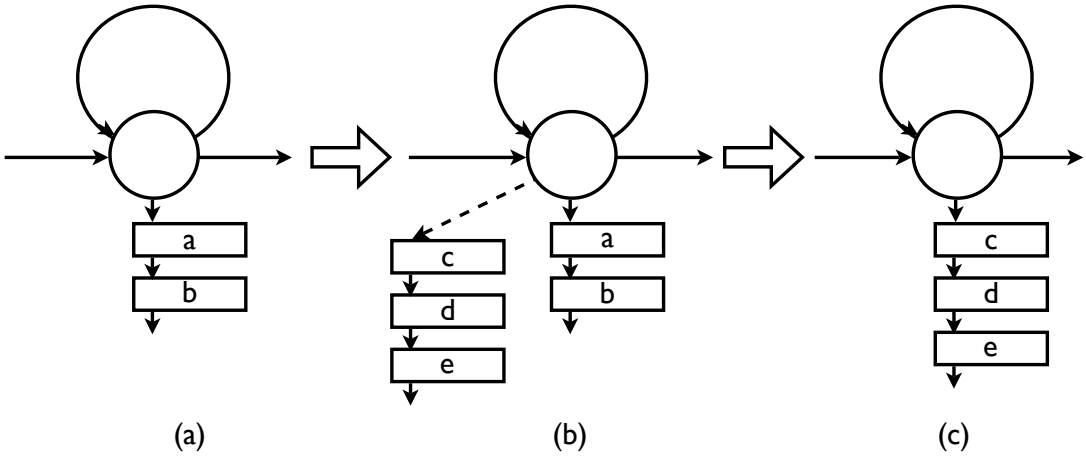


Figure 4.6: Instance Lists

Although instance installation is deferred to epoch boundaries, processing of new events for instances reaching final states is *not* deferred. Client notification, as well as insertion of new events into the PQ to support resubscription, both happen immediately. This scheme is a simple and efficient way to handle simultaneity and resubscription. It also makes it possible to construct some powerful recursive queries (arguably a good thing), or to construct an automaton that can loop forever in an epoch, generating unboundedly many events with the same detection timestamp (a bad thing). Fortunately, automata generated from Cayuga algebra expressions are guaranteed not to exhibit this bad behavior.

Evaluation and Indexing

We can now describe the flow of processing for each incoming event. The system components involved are shown in Figure 4.7. Our design is specially driven by the invariants of Cayuga automata described in Section 4.3.

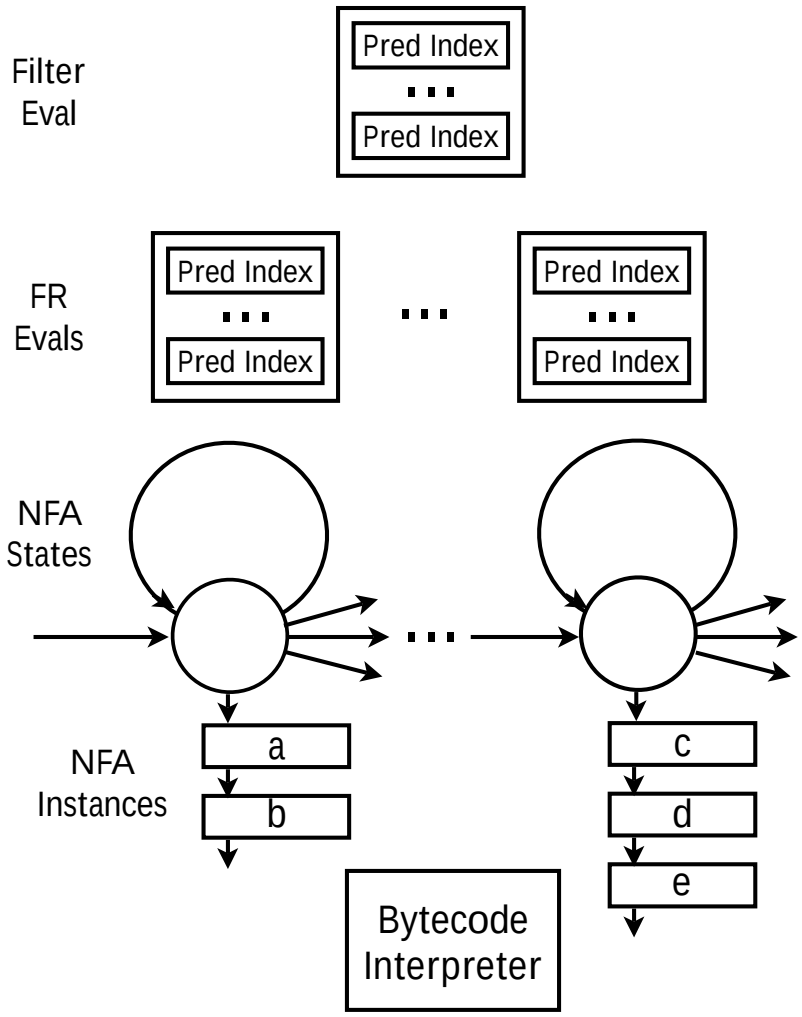


Figure 4.7: Query Evaluation Architecture

Event processing is done in two stages. In the first stage, the Filter Evaluator is invoked. As its name suggests, the Filter Evaluator evaluates filter edge predicates with respect to the current input event. It identifies the set of *affected* instances and the states they are associated with. (Recall from Section 4.3 that an instance is *affected* if it does not traverse its filter edge.) All affected instances will be marked for deletion at the end of the current epoch.

In the second processing stage, for each affected state marked by the Filter Evaluator,

we invoke the Forward-Rebind Evaluator. The FR Evaluator evaluates all the FR edge predicates of a state, and creates new instances under the destination state of each edge if its predicate is satisfied. Whenever an instance is created under a final state, an output event is generated, which is inserted on the PQ for resubscription and/or sent to the appropriate Client Notifiers.

In our implementation we convert edge predicates to Disjunctive Normal Form and treat the top-level disjuncts separately. Thus, for the remainder of this discussion we assume each predicate is a conjunction of atomic predicates. We classify atomic predicates as either *static* or *dynamic*. A static atomic predicate compares an event attribute to a constant. Any other atomic predicate is considered dynamic.

In a straightforward implementation of a Filter Evaluator with no indexing, each input event would need to be checked against the static filter atomic predicates of all the states; then the dynamic filter atomic predicates would need to be checked for all instances present at the nodes whose static filter components were satisfied, thereby identifying the affected instances and nodes. All these predicate evaluations would be performed by the Bytecode Interpreter.

Similarly, in an unindexed implementation of an FR Evaluator each input event would need to be checked against the static atomic predicates of every FR edge of every affected state; then, for those FR edges whose static components were satisfied, the dynamic atomic predicates would need to be checked for all affected instances at the associated node.

Performance is greatly improved by the use of indexing. Our current indexing structure addresses Static and Dynamic equality Atomic predicates. Each Conjunctive Clause is segregated into four components: Indexed Static, Unindexed Static, Indexed

Dynamic, and Unindexed Dynamic.

Static predicate indices for filter predicates are maintained at the global level to efficiently identify (a superset of) the set of affected States. Static predicate indices for FR predicates could be maintained at the global level (eliminating duplicate evaluation of indexed static atomic predicates) or on a per-node basis (eliminating unnecessary evaluation for unaffected nodes). Either choices could be viable, depending on workload; we decided in favor of the second. Dynamic predicate indices for filter predicates are maintained on a per-node basis, since they index node instances rather than the global constants in the static filter predicate, and each index is associated with a particular filter predicate clause. We currently maintain only one dynamic predicate index per node to avoid excessive maintenance overhead as instances are added and deleted at the end of each epoch.

Upon event arrival, the following process is carried out for filter predicate evaluation

1. The filter evaluator probes the relevant indices to enumerate which nodes have the static indexed component of one of their filter predicate clauses satisfied. (The clause number is also output). The static unindexed component is evaluated for these clauses, and if is is satisfied we move to step 2.
2. Once the node and clause numbers have been identified, the corresponding dynamic filter predicate index is probed to get the set of instances that satisfy the indexed dynamic component of that clause.
3. For each such instance, the unindexed dynamic component is evaluated by the interpreter to output the set of affected instances at that node.

For Nodes that do not have filter predicate indexing enabled, the index output degenerates to a sequential list consisting of all instances present at the node, unindexed

predicate evaluation then proceeds on this list. The process ultimately creates a list of affected nodes, each containing a list of affected instances of its own. The FREvaluator then probes the FR indices associated with each affected node to output the FR Edges having the static component of one of their clauses satisfied (the clause number is also output). Dynamic FR indices are currently not supported in our system. Since the number of affected nodes/instances is generally small, we believe the performance advantage would be overcome by the maintenance overhead. The unindexed component is evaluated on the node's affected instances to finally identify (edge, instance) pairs. New instances are created and added to pending instance lists to effect the state transition.

4.4.4 Client Notifiers

Client Notifier threads (CNs) are roughly analogous to ERs: ERs receive events from external streams, while CNs send event notifications to subscribing clients. There is one CN per connected Cayuga client. Whenever the Query Engine detects that a query has produced a match (i.e., an automaton instance reaches a final state), the QE sends the corresponding event to all the CNs representing the subscribing clients for that event. Each CN serializes the event appropriately and delivers it to the client.

4.4.5 Memory Management

Garbage Collector

During Cayuga event processing, large string bodies and objects of complex user-defined data types are created, copied and destroyed frequently. To reduce the space and time overhead for such objects, we share the objects as much as possible and rely

on a custom memory management scheme to reclaim them. In this section we describe the memory manager and discuss some of our design decisions.

The principle dynamic data structures in Cayuga are associated with events and automaton instances. Here we discuss the treatment of automaton instances; the treatment of events is similar, and in fact most of the code is shared.

At the top level, automaton instances are allocated and freed manually. Scalar data resides in each instance, while string bodies and complex data structures reside in a garbage-collected heap and are always shared among instances. Note this implies that the size of an instance is determined by the schema of the corresponding automaton state, and instances are fairly small. Copying an instance is done by "shallow copy" – only the fixed-length instance object is copied, increasing the degree of sharing of the sub-objects.

To manage instance objects, we use a simple size-segregated collection of free pools.

To manage complex shared objects, we initially considered a C++ "smart pointer" implementation based on reference counts. Apart from reference counting's inability to deal with cyclic structures – a restriction we could probably have lived with for our application – this approach has considerable elegance. However, Cayuga's nondeterministic state transitions require frequent copying and deletion of instances, each involving reference count manipulation and possibly synchronization overhead as well. So we implemented a garbage-collected heap for shared objects. This allows us to do a shallow copy of an instance with no locking or reference count manipulation – a simple and efficient block move operation is safe.

Our GC design uses some familiar techniques in a somewhat unusual combination driven by the needs of our application. Discussion of most of these techniques can be

found in [94].

First, we remark that in the absence of pathologic query selectivities Cayuga's object lifetime distribution is highly bimodal. Most automaton instances fail filter predicates and die very early. A few instances, such as those associated with a long-running μ operator, live for a very long time. But hardly any instances have "medium" lifetimes. Because of this observation, we chose a simple generational scheme with two generations [9]. The first generation uses copying garbage collection [8]; objects that survive more than a few copies are "promoted" to the second generation, which uses non-copying collection.

Although copying collection is somewhat controversial [20, 83], Cayuga's object lifetimes are sufficiently short and skewed that we believe copying should have a considerable performance advantage. With a copying GC, object allocation is essentially free – basically, just incrementing a limit pointer – and the cost of a collection is linear in the number of bytes of live data, but is *independent* of the number of bytes reclaimed. Thus, if the vast majority of allocated objects do not survive until the next garbage collection, the total cost of allocation using a copying collector can be lower than the cost of manual memory management using malloc() and free() [8].

To avoid the need to update all client reference variables when an object is copied, we use a handle-based design similar to some Java VMs. Our handle space data structure supports unit-cost reclamation of the entire set of dead handles during a reclamation, preserving the asymptotic cost of the copying collection algorithm.

We now briefly discuss issues of root finding and concurrency. Logically, the first step of a garbage collection is to find the values of all "root variables" – program variables that might contain references to objects in the heap. The collector then traces

through the heap, identifying all objects that are transitively reachable from any of the roots. In a multi-threaded system like Cayuga, the GC must either stop all other threads (a "stop-the-world" collector) or be prepared to operate correctly in the presence of concurrent changes to the heap made by other threads.

In general, these tasks represent considerable development effort. Precise root finding requires either (non-portable) compiler support or (error-prone) application support. Stopping the world is also non-portable, and is undesirable in any case; while concurrent collection is extremely delicate and error-prone [refs].

We address these issues by exploiting our application design, in which the majority of work is done by a single Query Engine thread. Most GC systems collect as a side-effect of object allocation. The Cayuga GC collects only when explicitly invoked by the Engine thread. This happens only at "GC-safe" points in the Engine code[2]. At such points, all of the Engine's live data is guaranteed to be reachable from (1) events waiting to be processed in the PQ, or (2) automaton instances contained in the query automata data structure. These are the only Engine root variables that are needed, and enumerating them requires only a few lines of code.

Of course, this approach guarantees only that the Engine thread is at a GC-safe point when a collection occurs. What about the ER and CN threads? We guarantee that *all* points in such threads are GC-safe by requiring them to access the heap using a stylized API similar to a Java "native methods" API. This API is somewhat less convenient than the direct API used in Engine code; but use of the heap by the ER and CN threads is simple enough that the approach is tolerable.

Finally, note that an allocation request may ask for more space than is currently

[2]In fact, Cayuga contains only one call to the collector, at the top of the Query Engine's main loop. Whether the call actually does a collection, or just returns immediately, is a policy decision internal to the GC.

available in the heap. The usual response in this situation is to invoke garbage collection, and to fail only if the collection does not produce sufficient free space. However, we have explicitly ruled out invoking garbage collection as a side effect of an allocation request. Fortunately, this is not a serious issue in our two-generation design. When insufficient memory is available in the first-generation heap to satisfy a request, we immediately "promote" the request to the second generation, which uses the system allocator and does not fail. [3]

Internal String Table

The Internal String Table is a component that manages read-only string objects stored (internalized) in the Cayuga Heap. The purpose of the Internal String Table is to ensure that there is at most one copy of any string value in the heap – if multiple clients internalize the same string value, the system returns identical references to the same shared string body in the heap. This has two beneficial effects:

- It reduces space consumption. If the same string appears in multiple input events or automaton instances, storage will be shared even if the events or instances were constructed independently.
- It "canonicalizes" the strings, enabling equality test to be performed by a single pointer comparison rather than byte-by-byte character comparison. For certain workloads, in which successful string comparisons occur frequently, this can yield significant speedups.

The "obvious" implementation for the Internal String Table is to place all internalized strings in a hash table, eliminating duplicates. Unfortunately, this implementation does

[3] More precisely, it does not fail recoverably.

not work, because it makes no provision for ever reclaiming an internalized string object s even if no client holds a reference to it – the Internal String Table *itself* holds a reference to s that keeps it from being reclaimed. Thus, the Internal String Table can grow without bound even if Cayuga's space requirements *at any particular time* are modest.

A Canonical String Table like ours is not a new idea; solutions to the unbounded growth problem typically involve some form of "weak references," a GC feature which, while still slightly arcane, dates back to the 1980's and is sufficiently mainstream to exist in Java and C-#. Informally, a weak reference to an object enables a client to use that object, but does not prevent the object from being reclaimed. The Cayuga GC provides a very simple form of weak reference: when all ordinary references to an object disappear, the object is reclaimed, and all remaining weak references to the object are atomically cleared to **null**. With this feature, we can easily implement the Internal String Table as a hash table whose buckets contain weak references to canonical string objects; code in the hash table maintenance methods lazily deletes **null** references from the table.

4.5 MQO Techniques in the Cayuga Query Engine

The Cayuga algebra and automaton model are designed to be amenable to multi-query optimization. An obvious optimization is to merge equivalent states that occur in several automata. This is the approach taken by YFilter [31]. The result of the merging process is a DAG with a single start node. In the following we first focus on implementation challenges that are unique to Cayuga. For this discussion we need some additional notation. Afterwards, a more general version of state merging will be introduced in Section 2. Section 4.5.6 describes the system architecture and the data flow in Cayuga.

4.5.1 Notation

A *static predicate* is a conjunction of atomic predicates that compare attribute values of the incoming event to constants, e.g., `Name` $=$ IBM $\wedge$ `Price` $>$ 10. A *dynamic predicate* (or *parameterized* predicate) is a conjunction of atomic predicates of the form ATT_1 relop ATT_2, which compares an attribute value of the incoming event with an attribute of an earlier event. An example is θ_2 in Example S3.

For ease of exposition, in the following discussion we assume that each predicate is a conjunction of atomic predicates. Our techniques can be easily generalized to arbitrary boolean combinations of atomic predicates by requiring that predicates be supplied in disjunctive normal form (DNF), a disjunction of conjunctions of atomic predicates. Each conjunction P can be rewritten as $P = \bigwedge_i \text{ATT}_i$ relop $\text{CONST}_i \wedge \bigwedge_j \text{ATT}_j$ relop ATT_{k_j}. We refer to $\bigwedge_i \text{ATT}_i$ relop CONST_i and $\bigwedge_j \text{ATT}_j$ relop ATT_{k_j} as the *static* and *dynamic* parts of P, respectively. If either part is empty, it is equivalent to TRUE.

A node of an automaton is *active* if there are automaton instances at the node. For each incoming event, an automaton instance is *unaffected* if that event makes the instance traverse its *filter* edge; otherwise it is *affected*. For example, in Example S2 the filter condition θ_1 ensures that after matching the high-price IBM quote, the corresponding instance of the automaton will be affected only by MSFT quotes and can safely ignore quotes for other companies.

4.5.2 Design Challenges

Effective multi-query optimization for Cayuga's stateful parameterized subscriptions must meet three crucial challenges. First, evaluation of Cayuga's subscriptions is driven

by edge predicates being satisfied (or not) for an incoming event. The number of active automaton instances and the number of edges that each instance could potentially traverse can be very large. Hence, evaluating all these edge predicates for each incoming event is not feasible. So we need to index the predicates, which is the classic pub/sub matching problem. Second, besides the static predicates handled by traditional pub/sub systems, Cayuga also needs to deal with dynamic predicates. This problem has not been studied in traditional pub/sub systems. Finally, although the *total* number of automaton instances can be very large at any time, the number of instances *affected* by an event is typically orders of magnitude lower. In the stock monitoring application, for example, a subscription that matches a sequence of IBM prices can ignore events for any other company. So we need an index that enables us to identify the affected instances quickly.

Observe that an instance is affected iff it cannot traverse the filter edge of its state (i.e., its filter predicate is satisfied). Therefore the problem of identifying affected instances (third challenge) is the same as the problem of efficiently finding predicates that are satisfied by an incoming event (challenge 1).

While we can use standard data structures for indexing static predicates, it is not obvious how to index dynamic ones (challenge 2). We propose two general approaches: (1) dynamic predicates are handled like static predicates once the parameter values are known, and (2) dynamic predicates are not indexed. The first approach is based on the observation that for an instance in automaton state X, all the parameters on the outgoing edges of state X are already bound by that instance. For example, in Example S3, assume the automaton advances to the first state on an incoming stock quote for IBM. Now the name parameter ($S_1.\texttt{Name}$ in θ_1) is bound to IBM, and hence θ_1 will check if the name attribute of later stock quotes is equal to IBM. At this time the corresponding predicate $S_2.\texttt{Name} =$ IBM can be inserted into a (pub/sub) index. There is an obvious

tradeoff with this approach: if we index the dynamic predicates, index maintenance becomes much more expensive than for the second approach. On the other hand, if we index only the static predicates the index will be less selective and might produce false positives, in particular those predicates whose static part is satisfied, but whose dynamic part is not.

In the following sections, we describe our solutions to challenge 2 for the case of indexing filter predicates and FR predicates (predicates on forward or rebind edges) respectively.

4.5.3 AN-index and AI-index

The goal of these indexes is to efficiently identify the instances that are affected by an incoming event. To do so, we index each instance by the filter predicate of its current state. More precisely, the index takes the filter predicate as the key and the corresponding instance as the value. We implement this index with a two-level scheme. The first level index only works on the static part of filter predicates. We refer to it as the *Active Node Index*, (AN-Index), since it essentially returns all the automaton instances of those active nodes on which the static parts of filter predicates are satisfied. Then, for each such node, the second level index, called the *Active Instance Index* (AI-Index), is used to further prune the candidate set of affected instances by indexing the dynamic part of the filter predicates.

One reason for this separation is that it enables us to leverage existing data structures. For the fairly static AN-index, we can use pub/sub technology like Le Subscribe [33]. However, to keep index maintenance costs in the second level low, the AI-indexes are simple hash tables. Hence only equality predicates are indexed. This nevertheless proves

to be a very useful feature for supporting parameterized atomic predicates like $\texttt{Name} = S.\texttt{Name}$, which simulates a grouping by Name and essentially has the same effect as the frequently-used "partition-by" window feature in CQL [67]. The two-level approach also simplifies data structure optimizations. If the system determines that for one of the AI-indexes the maintenance overhead exceeds the savings from improved selectivity, this AI-index can be disabled without affecting the use of the first level index.

4.5.4 FR-Index

Knowing the instances affected by an incoming event is not sufficient. We also have to determine which forward and rebind edges these instances will traverse. Traversing an FR edge modifies instance bindings, affecting the instance content; if no edge can be traversed, the instance is affected by being deleted. A second pub/sub-style index, called the *FR-Index*, is used in Cayuga to index the static part of the FR predicates. Since all FR predicates are conjunctions, after using FR-Index, we still need to eliminate false hits by post-processing those instances whose static predicates are satisfied by evaluating their dynamic predicates.

Here we do not index the dynamic part of each FR predicate, because for each incoming event, only the affected instances will need to have their FR predicates further evaluated. This leads to a much lower benefit-cost ratio compared to the problem of finding affected instances.

Figure 4.8 illustrates the relationship between the different indices with respect to how the search space of instances is pruned. AN-Index and AI-Index identify affected instances efficiently, while FR-Index evaluates the static part of FR predicates of each instance so that a decision of whether to advance or drop the instance can be made

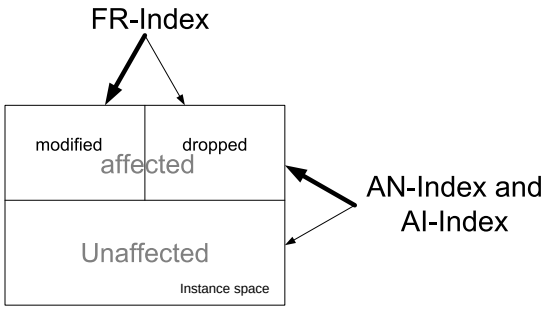


Figure 4.8: Instance Search Space

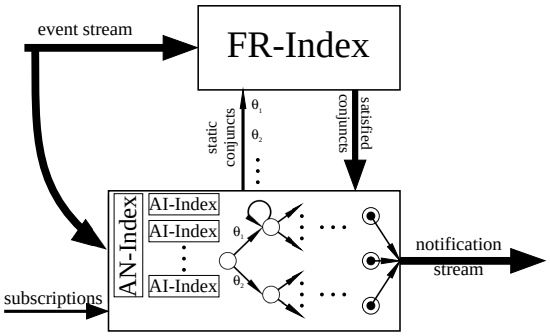


Figure 4.9: Cayuga architecture

quickly.

4.5.5 Merging Automata

Our automata have a structure similar to the automata of YFilter [31] or linear finite automata in general. Hence we can use a similar procedure for merging common prefixes of different automata. Our procedure is slightly more general, since the union operator creates a DAG (directed acyclic graph), rather than a tree, and since there is a greater variety of edge types and edge labels. The final result of the merging process is a DAG of all automaton states.

Formally, the merging of automata is based on the following notion of *equivalent states*.

Definition 2 *Let* n *and* m *denote automaton nodes (states), and* E_n *and* E_m *denote the sets of edges entering* n *and* m *respectively. We define a nested sequence*$\{\equiv_i \mid i = 0, 1, \ldots\}$ *of equivalence relations on states as follows.*

- $n \equiv_0 m$ *for all* n, m.

- $n \equiv_{i+1} m$ *if and only if there exists a bijection* $\sim$ *between the entering edge sets* E_n *and* E_m *such that for each mapped pair* $e_n \sim e_m$
  - e_n *and* e_m *have identical edge labels, and*
  - $e_n.source \equiv_i e_m.source$

States n *and* m *are* equivalent*, written* $n \equiv m$*, if and only if* $n \equiv_i m$ *for all* i*.*

It is possible to compute $\equiv$ using a slight generalization of the traditional techniques for state minimization of finite automata [46]. However, our current implementation takes a simpler approach, merging only *prefixes* of paths of equivalent automaton states, as in [31].

We can identify equivalent nodes efficiently by computing a hash signature of the set of predicates for each incoming edge, and using this signature to prune the search space. This is fairly straightforward and not discussed here.

4.5.6 Query Engine Architecture and Data Flow

The overall system architecture of Cayuga is shown in Figure 4.9. Its core component is the *State Machine Manager*, which manages the merged query DAG and the automaton instances at the nodes. It also maintains the AN-Index and AI-Index. Outside the State Machine Manager, there is the FR-Index.

Cayuga needs to handle two types of updates—insertion/deletion of subscriptions and arrival of input events. A new query is inserted by first merging it into the query DAG in the State Machine Manager.This is shown in Figure 4.10 for a simple example query. For simplicity we assume that only the start states can be merged. Then, for

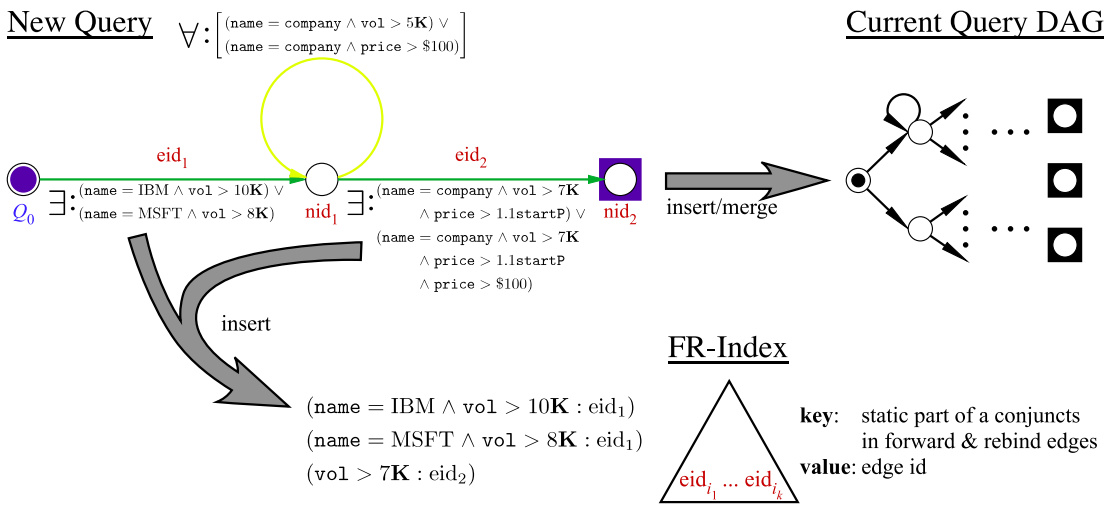


Figure 4.10: Insertion of a Query

each forward and each rebind edge, an entry is added into FR-index for the static part of the edge predicate. This entry has the static part of the conjunction as the key and the ID of the edge as the value. In the example, each of the two conjuncts of edge eid_1 results in a separate entry.AN-index and AI-index are not affected, because they maintain only active nodes and instances. When the query is deleted, the insertion process is simply reversed. Only nodes and edges that are not shared with other subscriptions are physically removed from the DAG.

Incoming events are sent to both the State Machine Manager and the FR-index. Figure 4.11 illustrates how an event is processed right after the new query was inserted. In the example we omit the timestamp attributes for readability. Probing the FR-index produces a set of edge IDs as the result. This is the set of edges whose static predicate parts are satisfied by the event. Note the index returns both eid_1 and eid_2, based on their static predicate parts. For efficiency during later probing, the resulting set of edge IDs is stored in a hash table. At the State Machine Manager, first AN-index is probed. It returns the set of active nodes whose filter edges are not traversed, based on the static parts of the filter predicate conjuncts. This immediately prunes a large number of active nodes,

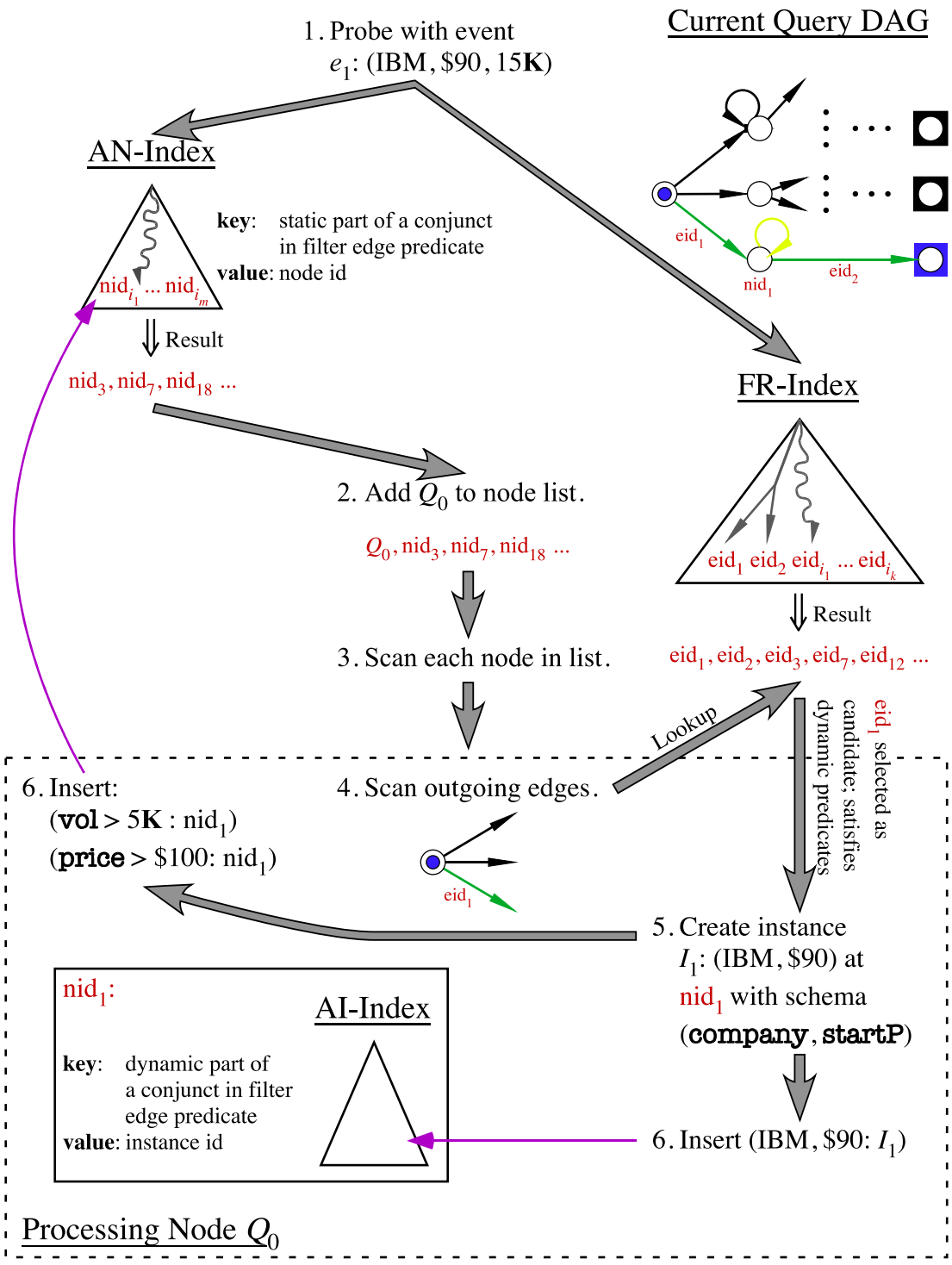


Figure 4.11: Processing First Event

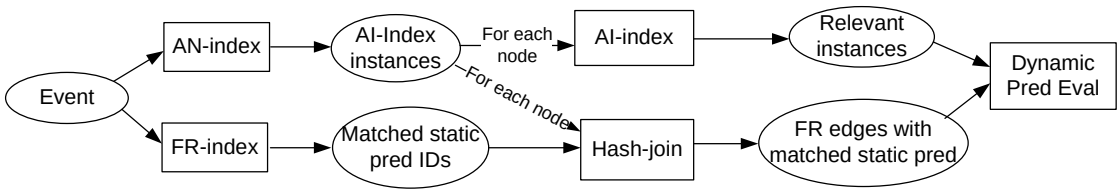


Figure 4.12: Event Processing Diagram

whose instances all traverse the filter edge. For each node in the result list of AN-index, the system determines which instances at the node are affected by the event by using the corresponding AI-index. If there is a hit, the corresponding edge is a *candidate* for a traversal. Recall the FR-index only uses the static parts of edge predicates; therefore, to eliminate false positives, we must test whether a candidate also satisfies the *dynamic* parts of the edge conjunctions. If it does, we create a modify the current automaton instance by extending the event bindings and advance it to the destination node.

The diagram in Figure 4.12 summarizes the Cayuga event processing steps. On arrival of an event, the following happens:

1. FR-index generates a set of IDs of the satisfied static predicates on FR edges.
2. AN-index returns a set of AI-Index instances.
3. For each AI-Index instance in the set we do the following. We first obtain from this AI-index the set of relevant instances for which the dynamic equality predicate of the filter condition is satisfied. For each of these instances the remaining dynamic atomic predicates of the filter edge are evaluated. This gives us the set of affected instances. Then we determine for each affected instance the candidates of satisfied FR edges by intersecting the output of FR-index with the set of IDs of the static FR predicates associated with the current node, followed by an evaluation of the dynamic parts of FR predicates whose static parts are satisfied.

4.6 Performance Evaluation

We built an initial prototype implementation of Cayuga in C++. For standard data structures such as hash indices and lists we relied on the C++ Standard Library implementations. We believe that using specifically tailored implementations would lead to a considerable gain in system performance. However, even with the current prototype implementation we show that, with no more than a standard, off-the-shelf PC, we can process thousands of events per second, for hundreds of thousands of concurrently active stateful subscriptions.

All experiments were run on a 3 GHz Pentium 4 PC with 1 GB of RAM and 512 KB cache. The operating system is Red Hat Linux 9. We loaded the input stream into memory before starting the experiment to make sure that the input tuples are delivered at least as fast as our system can process them. We measured the *total* runtime for matching all incoming events with all subscriptions in the system. For each experiment we perform several runs, clearing the cache between runs. As the standard deviation in all experimental runs was well below 1%, we therefore only report averages and omit error bars from the graphs.

4.6.1 Technical Benchmark

To test the overall efficiency of Cayuga and measure the evaluation cost of the different operators of our algebra, we designed a synthetic technical benchmark motivated by the stock application, but more complex to provide flexibility in subjecting our system to a stress test. We use a stream with eight data attributes: four discrete attributes (e.g. company name) and four continuous attributes (e.g. stock price). The parameters for

Table 4.1: Parameters (default values)

Variable	**Value**
Number of events	100,000
Number of attributes per event	8
Number of discrete attributes	4
Number of continuous attributes	4
Number of subscriptions	200,000
Number of atomic predicates (discrete + continuous)	2 + 2
Domain size of discrete attribute	100
Number of distinct ranges that can be selected for inequality predicates	25
Selectivity of atomic inequality predicate	0.7
Number of steps per sequence query	3
Zipf parameter, first step ($zipf_1$)	1
Zipf parameter, second step ($zipf_2$)	1
Zipf parameter, third step ($zipf_3$)	0.8
Duration constant (t)	20

generating the stream and the associated subscriptions are shown in Table 4.6.1.

We generated subscriptions according to five different templates: `LinearStat`, `LinearDyn`, `Filter`, `NonDeterministic`, and `NonDeterministicAgg`. All subscriptions are over a single input stream S; We use S to refer to an appropriately renamed occurrence of S in the algebraic expression.

`LinearStat` subscriptions define simple sequential patterns of three consecutive events, expressed as

$$\sigma_{\theta_3}(\sigma_{\theta_2}(\sigma_{\theta_1}(S_1)\,;S_2)\,;S_3)$$

in our algebra. Essentially, this query looks at any three consecutive events in the stream, and outputs the concatenated result if all of the three selections are satisfied. For example, if such a template were applied to our stock stream example, then our template might generate the following subscription.

Example 20 *Notify me when there are three consecutive stock quotes representing IBM below $10, followed by IBM above $15, and finally IBM below $15.*

As our input stream is not the stock stream, but a synthetic stream of eight attributes, the θ_i are conjuncts of four *static* atomic predicates: two equality predicates on two of the discrete attributes, and two inequality predicates on two of the continuous attributes. One of the discrete attributes, ATT, is designated as the *primary attribute* of the query. This attribute is guaranteed to appear in all three of the θ_i, and to select exactly the same value for each formula. The `Name` attribute in Example 20 is an example of such an attribute, as it is assigned to IBM in each case. As all of the formula select the same value, we refer to the predicate ATT $=$ CONST as the *primary predicate* of the query.

Attributes and their values are selected independently, using $zipf_1$ to select attributes and $zipf_i$ to select the value for θ_i. This setup is motivated by practical scenarios where user preferences typically follow a skewed (often Zipf) distribution. By adjusting the Zipf parameter, we can control the similarity of the different subscriptions.

To test the overhead of evaluating *parameterized* predicates in Cayuga, we designed the `LinearDyn` template based on `LinearStat` as follows.

$$\sigma_{\theta_3}(\sigma_{\theta_2}(\sigma_{\theta_1}(S_1)\,;S_2)\,;S_3)$$

The difference between this template and `LinearStat` is that θ_2 and θ_3 now have an additional *parameterized* atomic predicate. An example of such a predicate from our

stock stream would be the requirement that the stock price from the second quote is 1% above the price of the original quote.

The overhead of evaluating filter predicates is measured with the `Filter` template

$$\sigma_{\theta_3}(\sigma_{\theta_2}(\sigma_{\theta_1}(S_1) \underset{\theta_4}{;} S_2) \underset{\theta_5}{;} S_3)$$

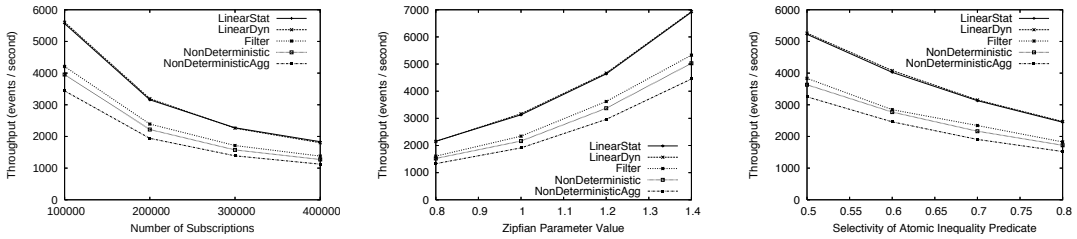
In this template, θ_1, θ_2, θ_3 are all selected in the same way as for `LinearStat`. On the other hand, θ_4 is a filter formula of the form $\text{DUR} \leq t \wedge S_2.\text{ATT} = \text{CONST}$, where t is as shown in Table 4.6.1 and $S_2.\text{ATT} = \text{CONST}$ is the primary predicate of the query in `LinearStat`. θ_4 relaxes the selectivity of the original `LinearStat` query by allowing intermediate non-matching events to be filtered out. To illustrate this idea with our stock stream example, suppose we took Example 20 and made θ_4 the filter predicate $\text{DUR} \leq 10\text{min} \wedge S_2.\texttt{Name} = \text{IBM}$. In this case, stock quotes of other companies that arrive between the first two IBM quotes would not lead to a failure of the pattern, as long as consecutive IBM quotes arrive within 10 minutes of each other. The second filter formula θ_5 is similar to θ_4; we merely replace $S_2.\text{ATT}$ with $S_3.\text{ATT}$.

The effect of non-determinism in our automata is measured by the `NonDeterministic` template

$$\sigma_{\theta_3} \circ \mu_{\text{ID},\theta_5}(\sigma_{\theta_2} \circ \mu_{\text{ID},\theta_4}(\sigma_{\theta_1}(S_1), S_2), S_3)$$

where ID is the identity unary operation. This query is much more powerful than the previous ones. An analogy using our example Example 20 would be a query that not only searches for patterns of *consecutive* IBM stock quotes, but one that can find *any n-tuple* of IBM stock quotes that satisfies the duration constraints and selection criteria θ_4 and θ_5, ignoring all stock quotes in between. Hence the output of this query will be a superset of the `Filter` query with exactly the same formulas θ_i.

Finally, template `NonDeterministicAgg` implements aggregation. It extends



(a) Throughput vs. number of subscriptions

(b) Throughput vs. Zipf skew

(c) Throughput vs. inequality selectivity

Figure 4.13: Throughput Measurements

`NonDeterministic` by computing the sum of the values of the continuous attributes, for the n events that satisfy the query pattern.

In processing these subscriptions, events were generated by uniformly selecting values for each of the eight attributes of the stream schema. We also examined skewed event distributions, but observed the same trends. Different distributions only affect results by changing the selectivity of the edge predicates. The same effect is achieved by adjusting the query constants, and so we did not investigate this further.

Results

Figure 4.13 illustrates the results of various throughput experiments. Figure 4.13(a) shows how the system throughput changes with the number of subscriptions. Even for 400K concurrently active subscriptions, throughput is well above 1000 events per second. As expected, the more complex the query workload, the lower the throughput, except for `LinearStat` and `LinearDyn`, which are almost identical because the cost of checking parameterized predicates is negligible compared to the other matching costs and the cost of maintaining the index structures.

Cayuga's high throughput is achieved for a challenging workload. Each event on average matches about 100 static predicates in the FR index. Furthermore, at any time, an average of 6000 to 16,000 nodes are active in the State Machine Manager, indicating that events satisfied a high percentage of the edge predicates. The high throughput was achieved because the index structures ensured that only about 40 to 120 of these active nodes had to be accessed per incoming event. Overall the `Filter` workload generated between 41 (100K subscriptions) and 171 (400K subscriptions) sequence matches, `NonDeterministic` and `NonDeterministicAgg` had a few more matches, and the linear workloads generated virtually no matches. This outcome is not surprising. Even though each single edge predicate by itself is not very selective, the overall pattern is very restrictive; the automata frequently advance to the first state, but only rarely reach the second or last state.

Note also that, despite the skewed query distribution, the merged query DAG is very large. For instance, before merging states the DAG for 100K subscriptions would have 300K nodes and edges. Our merged DAG still has about 215K nodes: 48K at level 1, 71K at level 2, and 96K at level 3. In the next result we show that a more skewed (hence more homogeneous) query workload can greatly improve throughput.

In Figure 4.13(b), we compare the effect of parameter $zipf_1$ on system performance. Lower skewness makes the subscriptions less similar, hence reduces the possibilities for state merging. This can be observed in the graph. Most of the performance difference is caused by the number of level 1 nodes in the query DAG, because that is where most activity takes place. For Zipf parameter 0.8, there are 101K nodes, while for Zipf parameter 1.4, there are 36K nodes. The overall number of matched subscriptions is virtually unaffected by the Zipf parameter, because there is no correlation between event values and query constants. This shows that state merging is effective when subscrip-

Table 4.2: Meaning of the curves

Mode Name	StateMerge	FR-Index	Instance Index
Cayuga	on	on	on
No State Merging	off	on	on
No FR-Index	on	off	on
No Instance Index	on	on	off
No MQO	off	off	off

tions follow a very skew distribution. However, by looking at the trend of curves in Figure 4.13(b), state merging becomes less important when the query distribution is less skew (e.g. zipfian value no greater than 1). This agrees with the result presented in the next section.

Finally, we examined the effect of edge predicate selectivity on the performance. Figure 4.13(c) shows how the throughput decreases when the inequality predicates on the continuous attributes select more values. Notice that the curve's slope is inverse quadratic, which is to be expected, as we are varying the selectivity of two predicates simultaneously.

4.6.2 Evaluation of MQO Techniques

In order to see the benefits of our MQO techniques, we run our system with different optimizations being turned on/off against the technical benchmark. We report only the result on `Filter` workload. Other results are similar.

Figure 4.14 shows the performance of Cayuga compared to four other system modes explained in Table 4.6.2. "Instance Index" corresponds to AN-Index + AI-Index. To

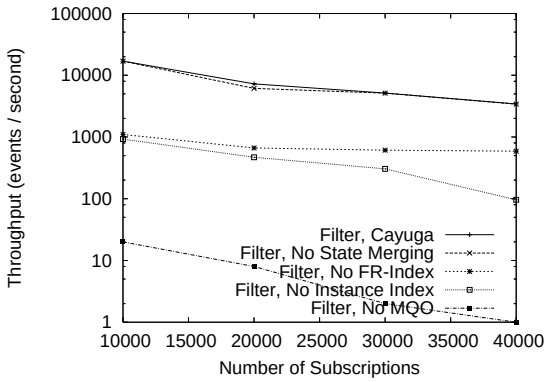


Figure 4.14: Effect of multi-query optimization

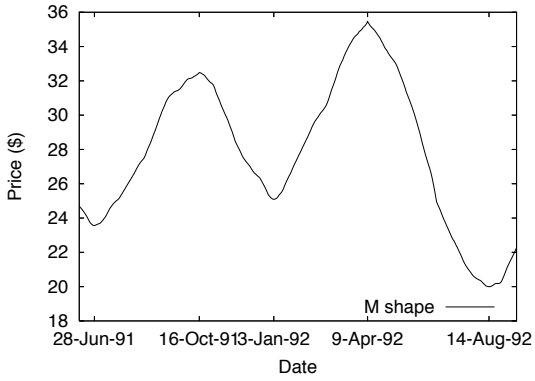


Figure 4.15: Double-Top pattern for Dell quotes

keep the runtime of the naive system manageable, we reduced the number of concurrently active subscriptions to 10K-40K, compared to 100K-400K in other experiments. Note that the y-axis is a *log scale*; hence with multi-query optimization the system is faster by a few orders of magnitude compared to that of a system without any of our MQO techniques.

It is clear from the graph that most of the performance gain comes from the indexing of FR predicates and instances, and not from merging automata states. This is true especially when the query workload is generated with a medium zipfian value, such as the default value 1.0 in our setup.

4.6.3 Experiment with Real Data

Stock Subscription

Full-fledged DSMSs are expressive enough to support extended pub/sub subscriptions, although they have only limited support for MQO and the query language based on SQL is not suitable for online event detection, as will be elaborated in Section 4.7.

```
# Stock stream
table : register stream Stock (time integer, name integer, price float);

# Add difference to previous stock price to each tuple
vquery : Rstream (Select S.time, S.name, S.price, (S.price − P.price) From Stock [Now] as S,
        Stock [Partition By P.name Rows 2] as P Where S.name = P.name and S.time > P.time);
vtable : register stream StockDiff (time integer, name integer, price float, pdiff float);

# Generate stream of extrema
vquery : Rstream (Select P.time, P.name, P.price, P.pdiff From StockDiff [Now] as S,
        StockDiff [Partition By P.name Rows 2] as P
        Where S.name = P.name and (S.pdiff * P.pdiff) < 0.0);
vtable : register stream Extrema (time integer, name integer, price float, pdiff float);

# Assign unique sequence numbers to extrema points
vquery : Select name, count(*) from Extrema Group By name;
vtable : register relation ExtremaCounter (name integer, seqNo integer);

# Attach sequence numbers to extrema
vquery : Rstream (Select E.name, E.price, E.pdiff, C.seqNo, C.seqNo − 1
        From Extrema [Now] as E, ExtremaCounter as C Where E.name = C.name);
vtable : register stream ExtremaSeq (name integer, price float, pdiff float,
        seq integer, prevSeq integer);

# State A: minimum
vquery : Select name, price, seq from ExtremaSeq Where pdiff < 0.0;
vtable : register relation stateA (name integer, price float, seq integer);

# State B: maximum, B > 1.2 A
vquery : Rstream (Select E.name, E.price, A.price, E.seq From ExtremaSeq [Now] as E,
        stateA as A Where E.name = A.name and E.prevSeq = A.seq and E.price > (A.price*1.2));
vtable : register relation stateB (name integer, bprice float, aprice float, seq integer);

# State C: minimum, 0.9 A < C < 1.1 A
vquery : Rstream (Select E.name, E.price, B.bprice, B.aprice, E.seq From ExtremaSeq [Now] as E,
        stateB as B Where E.name = B.name and E.prevSeq = B.seq and E.price > (B.aprice * 0.9)
        and E.price < (B.aprice * 1.1));
vtable : register relation stateC(name integer, cprice float, bprice float, aprice float, seq integer);

# State D: maximum, 0.9 B < C < 1.1 B
vquery : Rstream (Select E.name, E.price, C.cprice, C.bprice, C.aprice, E.seq
        From ExtremaSeq [Now] as E, stateC as C Where E.name = C.name and E.prevSeq = C.seq
        and E.price > (C.bprice * 0.9) and E.price < (C.bprice * 1.1));
vtable : register relation stateD (name integer, dprice float, cprice float, bprice float, aprice float,
        seq integer);

# The final query: D < A
query : Rstream (Select E.name, E.price, D.dprice, D.cprice, D.bprice, D.aprice
        From ExtremaSeq [Now] as E, stateD as D Where E.name = D.name and E.prevSeq = D.seq
        and E.price < D.aprice);
```

(a) Formulation in STREAM (CQL1)

```
# Stock stream
table : register stream Stock (time integer, name integer, price float);

# Generate stream of extrema
vquery : Istream (Select S2.time, S2.name, S2.price, (S2.price−S1.price)
From Stock [Partition By S1.name Rows 3] as S1,
    Stock [Partition By S2.name Rows 3] as S2,
    Stock [Partition By S3.name Rows 3] as S3
Where S1.name = S2.name and S2.name = S3.name and
    (S2.price−S1.price) * (S3.price−S2.price) < 0.0 and
    S1.time < S2.time and S2.time < S3.time);
vtable : register stream Extrema (time integer, name integer, price float, pdiff float);


# Assign unique sequence numbers to extrema points
vquery : Select name, count(*) from Extrema Group By name;
vtable : register relation ExtremaCounter (name integer, seqNo integer);

# Attach sequence numbers to extrema
vquery : Rstream (Select E.name, E.price, E.pdiff, C.seqNo, C.seqNo − 1
        From Extrema [Now] as E, ExtremaCounter as C Where E.name = C.name);
vtable : register stream ExtremaSeq (name integer, price float, pdiff float,
        seq integer, prevSeq integer);

# State A: minimum
vquery : Select name, price, seq from ExtremaSeq Where pdiff < 0.0;
vtable : register relation stateA (name integer, price float, seq integer);

# State B: maximum, B > 1.2 A
vquery : Rstream (Select E.name, E.price, A.price, E.seq From ExtremaSeq [Now] as E,
        stateA as A Where E.name = A.name and E.prevSeq = A.seq and E.price > (A.price*1.2));
vtable : register relation stateB (name integer, bprice float, aprice float, seq integer);

# State C: minimum, 0.9 A < C < 1.1 A
vquery : Rstream (Select E.name, E.price, B.bprice, B.aprice, E.seq From ExtremaSeq [Now] as E,
        stateB as B Where E.name = B.name and E.prevSeq = B.seq and E.price > (B.aprice * 0.9)
        and E.price < (B.aprice * 1.1));
vtable : register relation stateC(name integer, cprice float, bprice float, aprice float, seq integer);

# State D: maximum, 0.9 B < C < 1.1 B
vquery : Rstream (Select E.name, E.price, C.cprice, C.bprice, C.aprice, E.seq
        From ExtremaSeq [Now] as E, stateC as C Where E.name = C.name and E.prevSeq = C.seq
        and E.price > (C.bprice * 0.9) and E.price < (C.bprice * 1.1));
vtable : register relation stateD (name integer, dprice float, cprice float, bprice float, aprice float,
        seq integer);

# The final query: D < A
query : Rstream (Select E.name, E.price, D.dprice, D.cprice, D.bprice, D.aprice
        From ExtremaSeq [Now] as E, stateD as D Where E.name = D.name and E.prevSeq = D.seq
        and E.price < D.aprice);
```

(b) Formulation in STREAM (CQL2)

Figure 4.16: Double-Top query formulation

In this experiment, we compare Cayuga to the Stanford STREAM system, a general stream processing system with a relatively mature implementation. Since multi-query optimization has not been fully integrated into STREAM, we restrict the comparison to a single query at a time.

As for the query template, we use the well-known Double-Top (or M-shape) pattern used for stock analysis (see for instance `http://www.stockcharts.com`) that consists of five consecutive local extrema of stock prices, starting with a minimum at price A, rising monotonically to reach a maximum price B, such that $B \geq 1.2A$, then falling monotonically to reach a minimum C, which is within 10% of the starting price

A. Afterwards the stock rises monotonically to reach a new high of D, which is within 10% of the previous high B. Finally the stock falls monotonically to a new minimum E below A. Figure 4.15 shows an example of the pattern, found in a real sequence of stock closing prices.

The Double-Top query is naturally expressed in our algebra as a *linear-plus expression* with five μ operators (one for each monotonic sequence). It is essentially an extension to the `NonDeterministic` template in our technical benchmark, which contains only two μ operators.

It is possible to express this query in our algebra without the μ operator. While the resulting query is not linear-plus, we can implement it using the idea of resubscription from Section 3.3.4. To see how, first we create the following expression for detecting local maxima.

$$\sigma_{\theta_1}(\sigma_{\theta_2}(S_1 \underset{\theta_3}{;} S_2) \underset{\theta_4}{;} S_3)$$

Here θ_1 is $(S_2.\texttt{Price} > S_3.\texttt{Price})$, θ_2 is $(S_2.\texttt{Price} > S_1.\texttt{Price})$, θ_3 is $(S_2.\texttt{Name} = S_1.\texttt{Name})$, and θ_4 is $(S_3.\texttt{Name} = S_1.\texttt{Name})$. Note that local minima can be detected similarly. We then define stream E to be the union of all local minima generated by the above expression. Given this stream E of local minima, we now use resubscription to express Double-Top as an expression linear-plus in E. The expression is

$$\sigma_{\theta_1}(\sigma_{\theta_2}(\sigma_{\theta_3}(\sigma_{\theta_4}(E_1 \underset{\theta_5}{;} E_2) \underset{\theta_6}{;} E_3) \underset{\theta_7}{;} E_4) \underset{\theta_8}{;} E_5)$$

where the selection formula are as follows:

$$\begin{aligned}
\theta_1 &\equiv (E_5.\texttt{Price} \leq E_1.\texttt{Price}) \\
\theta_2 &\equiv (0.9E_2.\texttt{Price} \leq E_4.\texttt{Price} \leq 1.1E_2.\texttt{Price}) \\
\theta_3 &\equiv (0.9E_1.\texttt{Price} \leq E_3.\texttt{Price} \leq 1.1E_1.\texttt{Price}) \\
\theta_4 &\equiv (E_2.\texttt{Price} \geq 1.2 * E_1.\texttt{Price}) \\
\theta_5 &\equiv (E_2.\texttt{Name} = E_1.\texttt{Name}) \\
\theta_6 &\equiv (E_3.\texttt{Name} = E_1.\texttt{Name}) \\
\theta_7 &\equiv (E_4.\texttt{Name} = E_1.\texttt{Name}) \\
\theta_8 &\equiv (E_5.\texttt{Name} = E_1.\texttt{Name})
\end{aligned}$$

The two Cayuga formulations are referred to as `Mu Formulation` and `Resubscription` respectively in the experiment result.

STREAM's CQL query language lacks the μ operator. To efficiently find this pattern in CQL, the query is decomposed into manipulations on several levels of views (see Figure 4.16(a)). [4] Hence this implementation is is similar to the algebra expression with resubscription in our system. The STREAM implementation first computes a stream of "up" and "down" trends between consecutive quotes of the same stock. Then it detects local extrema in that stream. Finally, every sequence of five consecutive extrema is examined to determine whether the constraints on the price attribute are satisfied. Note that, while nicely optimized, this query had to be created manually, and it requires considerable expertise to craft the query in this way.

A more direct way to formulate this query in CQL, denoted as CQL2, is to use self-joins. This approach is shown in Figure 4.16(b). First a 3-way self-join is used to discover local extrema. Then, on the resulting stream of extrema, we find the actual pat-

[4] We would like to thank Arvind Arasu for crafting this CQL query formulation for us.

tern using a 5-way self-join. In order to properly quantify the performance degradation caused by a single n-way self-join, in CQL2 we express the first part of the query in the standard 3-way self-join fashion, but use the more efficient state-like expressions of CQL2 for the latter part.

Figure 4.17 shows the performance difference between the two equivalent formulations in Cayuga, and the two in STREAM. We run a single instance of the Double-Top query on a stream of 112,635 real daily closing stock prices for 24 different companies listed at the NYSE. The effect of different degrees of smoothing (length of window for computing a running average) is examined. Note that stronger smoothing reduces the number of local extrema, and hence benefits the resubscription and STREAM query formulation.

The `Mu Formulation` clearly outperforms the `Resubscription` formulation in Cayuga, as well as the two CQL formulations in STREAM. It is important to point out that the Cayuga `Resubscription` formulation and CQL1 perform almost identically, which should not be too surprising as the "state-based" spirit of their formulation is essentially identical too. However, CQL1 is essentially a hand-optimized version of the Double-Top query, whereas the more natural formulation of CQL2 (a declarative way to express sequence event composition with a multi-way join in CQL) had less than half of the throughput of CQL1, and a fourth of the throughput of the `Mu Formulation`. This supports our viewpoint that Cayuga is more suitable to process such extended pub/sub subscriptions that compose a sequence of events.

Table 4.3: Template Name and Description

`Stateless`: return all articles from website W with popularity $> X$.
`Concatenation`: return a series of 3 articles from website W with popularity $> X$.
`Parameterization`: return a series of 3 articles from website W on the same channel with increasing popularity.
`Iteration`: return a series of N articles from website W on the same channel with increasing popularity. N unbounded.

Subscriptions on RSS Feeds

We obtained RSS V2.0 feeds from 415 websites and preprocess them before feeding them into Cayuga. The preprocessor converts each RSS feed item into a Cayuga event by hashing the string values of the RSS fields to integers[5]. Some RSS fields such as <title> and <link> occur in each item, while others such as <author> are optional. To be able to pose interesting subscriptions, we augment the event schema with three additional attributes: website, channel, and popularity. The information of the first two attributes can be obtained directly from the feeds, while that of the last attribute is obtained through an external source that maintains the hit counts of these feeds. We sort the feed items by their publication date (<pubDate> field) and form an event stream of 26,623 events. The number of attribute/value pairs in each event varies from 6 to 11.

As for subscriptions, without the knowledge how a typical user would want to use our system, we composed four query templates shown in Table 4.6.3. To generate 10K to 40K subscriptions for each template, we randomly pick integer values to instantiate W and X. The domain sizes of W and X are respectively 415 and 100. The duration constraint of each query is fixed to be no more than 100 events.

[5]At the time when we conducted this experiment, the prototype implementation of Cayuga was not able to handle string comparison operations.

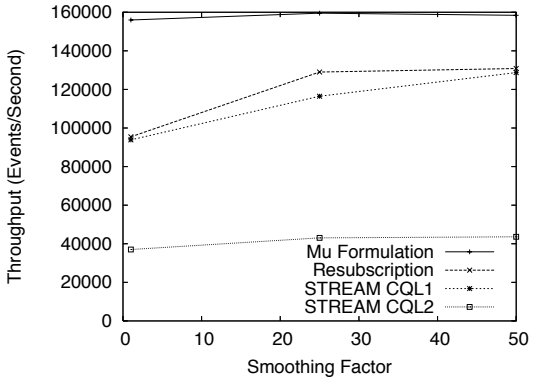


Figure 4.17: Comparison to STREAM

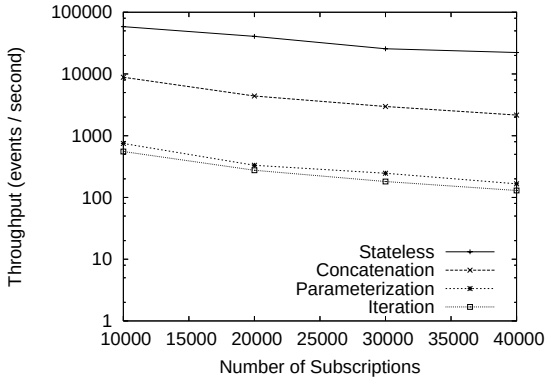


Figure 4.18: RSS Subscription

The result is shown in Figure 4.18. As we can see, The trade-off between query expressiveness and system throughput is well exhibited. However, even when processing 40K subscriptions of `Iteration` template, where thousands of witnesses are found and output, the system can still maintain a throughput of more than 100 events per second.

4.7 Related Work

To date, interest in building an online message brokering system has been spread across the expressiveness spectrum. At the low end of the spectrum lie pub/sub systems [6, 98, 33]. These systems sacrifice expressiveness to achieve high performance. For example, Le Subscribe [33] is a very high-performance scalable pub/sub system that performs aggressive multi-query optimization. Work in this area includes scalable trigger mechanisms [45, 74, 87].

There have also been quite a few systems for large-scale filtering of streaming XML documents [31, 23, 42, 40]. Their query languages usually are fragments of XPath, which is more expressive than pub/sub. However, XML filtering systems do not ad-

dress parameterization, and they cannot handle subscriptions across multiple XML documents. Automata are also a popular choice for many systems in this category [31, 42]. Our FR-Index can be potentially useful to YFilter, given that currently YFilter will have to sequentially evaluate all the structure predicates (usually equality comparison on string tags) on out-going edges for each active node to make non-deterministic state transitions.

Somewhat higher in the expressiveness spectrum is work from the Active Database community [93] on languages for specifying more complex event-condition-action rules. The composite event definition languages of SNOOP [22, 5] and ODE [36] are important representatives of this class. Both systems describe composite events in a formalism related to regular expressions, allowing events to be recognized using a non-deterministic finite automaton model. The automaton construction of [36] supports a limited form of parameterized composite events defined by equality constraints between attributes of primitive events. However, the semantics of some of the more expressive event languages is not well-defined [35, 100], and it is not clear how the different languages compare to each other in terms of expressiveness. In addition, the performance of event processing systems with very expressive query languages has not been explored in depth, especially in terms of scalability with the number of subscriptions. Our work can be viewed as extending this style of system with full support for parameterized composite events and support for aggregate subscriptions, focusing on multi-query optimization using a combination of state merging and indexing techniques.

There has also been some recent work on complex event processing, aiming at processing RFID streams [90, 95]. None of these systems provides strong support for multi-query optimization, a key ingredient for large-scale event processing. RCEDA [90] extends SNOOP, and supports temporal constraints on input events, as well as pseudo

events not generated by data sources (e.g., RFID sensors). The experimental result shows that RCEDA is scalable with the number of primitive events to process, but exhibits exponential behavior with the number of rules (queries) in the system. Multi-query optimizations are thus needed to improve the scalability of the system. SASE [95] supports novel language features such as negation and event detection overing a sliding window, and demonstrates significant performance gain in processing complex event queries compared to traditional data stream processing system TelegraphCQ. However, it has the following limitations. First, its data model prevents the system from supporting events with non-trivial interval, as well as simultaneous events (events detected at the same time). In comparison, the data model of Cayuga is sufficiently general to support both types of events. Second, the event (query) language of SASE is not composable, which restricts the set of queries expressible in the system. Observing the hierarchical nature of complex event composition, we believe composability is a must for an event language. Cayuga algebra is fully composable, and thus enables users to write non-linear-plus expression in a single query, as well as queries over the outputs of queries (view streams). Third, though negation is a novel feature propose by SASE, its evaluation strategy does not seem to be well investigated. For example, according to the description of SASE, to implement query `SEQ(A, !B, C)`, for each pair of A and C detected by the system, all events in between will have to be buffer to check the non-occurrence of B. This approach does not scale well with the window size of the query and the number of A, C pairs it produces, and the cost of buffer storage may become prohibitive in an online stream processing setting. Unfortunately there is no experimental result on negation to ease the concerns above.

Still higher in the spectrum, several groups are building systems with very expressive query languages [21, 67, 24, 3]. Sistla and Wolfson [82] describe an event definition and aggregation language based on Past Temporal Logic. The TREPLE language [66] is a

Datalog-based system with a precise formal specification; it extends the parameterized composite event specification language of EPL [65] with a powerful aggregation mechanism that is capable of explicit recursion. Perhaps the most powerful formal approach is STREAM's CQL query language [67], which extends SQL with support for window queries. Like SQL itself, CQL is declarative and admits of a formal specification [11]; and there are some initial results characterizing a sub-class of queries that can be computed with bounded memory [85, 10]. However, as we pointed out in the introduction, it is not clear whether SQL based languages with set semantics are suitable for real-time event detection and composition. Similar to SQL, the data model underlying these stream query languages is unordered, and so in order to pin-point the i-th tuple (in terms of temporal order) within a set of N tuples returned by a window operator, an N-way self-join with temporal constraints on these N tuples is required. A similarly powerful approach is represented by Aurora and Borealis [21, 3]. These two systems, however, use a procedural boxes-and-arrows paradigm which is much less amenable to formal specification in our style. Without formal semantics, it is hard to prove the correctness of query formulations, and opportunities for query rewrite/optimization in such systems are limited since many operator boxes are treated as black boxes. In [57] it is shown that SQL lacks expressive power for continuous queries on data streams, and Wang et al. extend SQL with features to support data mining and data streams [91].

There has also been some work in extending the expressiveness of pub/sub systems [59, 58]. However, [59] focuses on a distributed setting, and the degree of expressive power achieved by its query language is not as high as our algebra (e.g. no parameterization), and its implementation does not have MQO techniques other than state merging. There is no query language defined in [58], and the notion of a "stateful" subscription there is based on "state transition"; that is, when a regular (stateless) pub/sub subscription starts to be satisfied, or ceases to be satisfied.

Related to our implementation, Sellis [76] is one of the first to address general multi-query optimization in databases. Traditionally this is performed by sharing operators and query results [14, 21, 24, 54]. Our multi-query optimization is fundamentally different and aggressively exploits the relationship of our event algebra to automata.

Work on temporal and sequence database systems has emphasized static datasets instead of data streams [71, 78, 73]. In comparison, our approach treats events as having positive duration, an idea previously explored by the Knowledge Representation community [35]. In active databases [93, 87] and event-based processing we also want to match a large number of triggers with streams of incoming events [45, 74]; however, these approaches do not scale to the rates of the high-speed data streams that are the focus of this chapter.

4.8 Remarks

We believe that Cayuga incorporates interesting novel design decisions that will have impact on the community of researchers and practitioners that are currently working on building complex event systems. At Cornell, we currently have three ongoing deployments of Cayuga. We are collaborating with researchers from the Cornell Parker Center for Investment Research on real-time analysis and correlation of external data streams and stock data streams. We are also working on an integration with CorONA, a distributed high-performance publish-subscribe system for Web MicroNews running on PlanetLab (http://www.cs.cornell.edu/People/egs/beehive/corona/). Our vision is to devise an infrastructure for stateful monitoring of the Blogosphere. Our third deployment is an ongoing collaboration with system administrators at CTC, Cornell's high-performance computing center, on distributed infrastructure monitoring through OS

event log processing.

While our work here describes a working system that is currently under deployment, our users have started to demand other features. We are currently adding support for user defined types and functions in Cayuga; this also allows us to extend Cayuga to XML for deployment in service-oriented architectures. For an even more scalable architecture, we want to distribute the event processing task by setting up a hierarchy of Cayuga servers, where servers at higher levels resubscribe to the output of servers at lower levels.

CHAPTER 5

RULE BASED MULTI-QUERY OPTIMIZATION FRAMEWORK

5.1 Introduction

Query optimizers have been instrumental for the success of relational database technology. The cost difference between a good and a bad query plan can be several orders of magnitude. For data stream systems the stakes are even higher. Instead of one-shot queries in a relational DBMS, a stream system is processing *many continuous queries simultaneously*. These queries are active for long periods of time and they process massive streams in realtime as data is streaming in. Hence a poor query implementation choice can negatively affect system performance for the lifetime of this query.

The key to achieving good stream processing performance is to optimize the different queries together, rather than individually. In a stream query workload, it is often the case that multiple concurrently active queries can share state and computation. Query evaluation techniques that exploit this property are referred to as Multi-Query Optimization (MQO) techniques. The importance of MQO for stream processing is widely accepted and various stream MQO techniques have been proposed [33, 63, 43, 99, 54, 55, 29]. However, these techniques only apply to very specific queries or workload properties. For example, predicate indexing [33, 63] is taylored for a set of selection operators that all read the same input stream. In addition, it is particularly challenging to integrate the MQO techniques for CQL-style stream engines [11, 24], referred to as *Relational Engines (RE)*, and event pattern detection engines [30, 95], referred to as *Event Engines (EE)*. The former use an operator model similar to relational databases, while the latter implement queries with automata. A powerful stream query optimizer should be able to leverage all previously proposed MQO

approaches, and it has the potential to benefit from additional synergy created from the combination of these ideas in one system.

In this chapter, we propose a Rule-based MQO framework, called *RUMOR*. It is inspired by the classical Query Graph Model (QGM) used by RDBMSes [70], where query optimization techniques for *single* queries can be naturally modeled as transformation rules on query plans. This provides a modular and extensible framework for new optimization techniques to be developed and incorporated incrementally into the system. To support rule-based MQO, we have to extend the key abstractions that are used in a traditional RDBMS or stream system: physical operators, transformation rules, and streams.

We introduce a small number of carefully designed abstractions that together create a powerful MQO framework. In fact, RUMOR incorporates all previously proposed MQO techniques for stream processing. In particular, it successfully incorporates MQO techniques from both relational stream engines and automata-based event processing engines. Hence an additional benefit of RUMOR is that it enables the unification of these diverse camps of stream processing systems. Experimental results for our prototype implementation of RUMOR indicate that we can efficiently process a large number of CQL-syle relational stream queries, event processing queries, as well as *hybrid queries* involving query features from both types of query workloads.

RUMOR lays the foundations for multi-query optimizers (MQOptimizers) for data stream processing. It opens up opportunities for exciting future research on finding new rewrite rules and on extending the approach to cost-based MQOptimizers, incorporating ideas from the classical dynamic programming approach to cost-based *single* query optimization in RDBMSes [75].

Contributions and roadmap. Our contributions can be summarized as follows.

- We propose RUMOR, a rule-based MQO framework, which naturally extends the rule-based query optimization and query-plan-based processing model used by current RDBMSes and stream systems.
- We show how existing and new MQO techniques for relational stream engines and for event engines can be integrated into RUMOR. This is done by defining a small number of carefully designed abstractions.
- We implemented a prototype of RUMOR and present experimental results demonstrating the efficacy of our approach.

RUMOR integrates MQO techniques for REs and EEs. For ease of exposition, in Section 5.2 and 5.3, we interleave the description of RUMOR and that of how to integrate MQO techniques for REs in RUMOR. We then describe the integration of MQO techniques for EEs in Section 5.4. The experimental results are presented in Section 5.5. Finally, we survey related work in Section 5.6, and conclude in Section 5.7.

5.2 RUMOR: Part I

RUMOR incorporates three abstractions, respectively extending physical operators, transformation rules, and streams. For ease of exposition, in this section we introduce only the first two abstractions (Sections 5.2.2 and 5.2.3), and show how they can be used to express a set of interesting MQO techniques (Section 5.2.4). We describe the last abstraction in Section 5.3. We choose to present RUMOR in an intuitive way, accompanied by examples. The formalisms are omitted.

5.2.1 Background

We briefly review the related concepts in a relational query processing engine. A *logical query* is specified by a user through a query language such as CQL, which has well-specified semantics. A *query optimizer* reads a logical query as input, and produces a *physical query*, also known as a *query plan*, as the result of optimization. The optimization process involves the application of *transformation rules*, also known as rewrite rules, on the query plans. A transformation rule maps one query plan to another semantically equivalent plan (e.g. pushing selection below join). The query plan produced by the optimizer is executed by the *query engine* to produce results conforming to the logical query semantics. For this reason, we say the query plan *implements* its corresponding logical query.

In stream processing, for efficiency we want the query engine to process multiple queries together. We therefore extend the notion of a query plan to be one that implements *all* the currently active logical queries. A query plan is composed of *physical operators*, the basic scheduling and execution units in the engine. A physical operator *consumes* one or multiple input streams, and it *produces* one output stream. A physical operator is called the *consumer operator* of its input streams, and the *producer operator* of its output stream.

This chapter focuses on rewrite rules for query plans.

5.2.2 Physical Multi-Operator

In order to express and integrate a diverse set of MQO techniques, we need to understand the abstract nature of an MQO technique. Given an input query plan, an MQO technique

identifies opportunities for sharing, and it modifies parts of the query plan to exploit this opportunity. For example, if the query plan contains multiple selection operators reading the same input stream, the *predicate indexing* MQO technique shares work among them. It does so by indexing the selection predicates of the operators. For each incoming stream tuple this index is probed. It returns all satisfied predicates at a much lower cost than the naive strategy of evaluating each selection predicate individually one-by-one [33, 63].

To model a set of operators with shared computation, we propose an abstraction called *physical multi-operator* (or *m-op*). We say that an m-op *implements* a set of operators. An m-op is defined as follows. For every stream S, S is an input (resp. output) stream of the m-op, if and only if it is an input (resp. output) stream of one of the operators the m-op implements. The semantics of the m-op are defined as follows. Let t be an input tuple arriving in stream S. Then the m-op *conceptually* executes all its operators that have input stream S, and it writes the output produced for t by these operators to the corresponding output streams. The state of the m-op *conceptually* is a vector; each entry in the vector is equivalent to the state of one of the implemented operators if this operator was executed in isolation.

Notice that the definition of m-op semantics is based on the one-by-one execution of the implemented operators without sharing state. This defines the correct semantics, but obviously our goal is to find a more efficient m-op implementation that still guarantees the same input-output behavior as defined by the above semantics. Intuitively, the m-op consumes the set of input streams of the physical operators it implements, and it produces a corresponding set of output streams. The notion of consumer and producer operators for physical operators extends naturally to m-ops.

Clearly, the m-op abstraction generalizes the traditional physical operator abstrac-

tion. It therefore takes the place of a physical operator in RUMOR: A query plan is composed of a set of m-ops, and an m-op is the new scheduling and execution unit in the query engine. We illustrate the use of m-ops in the following example.

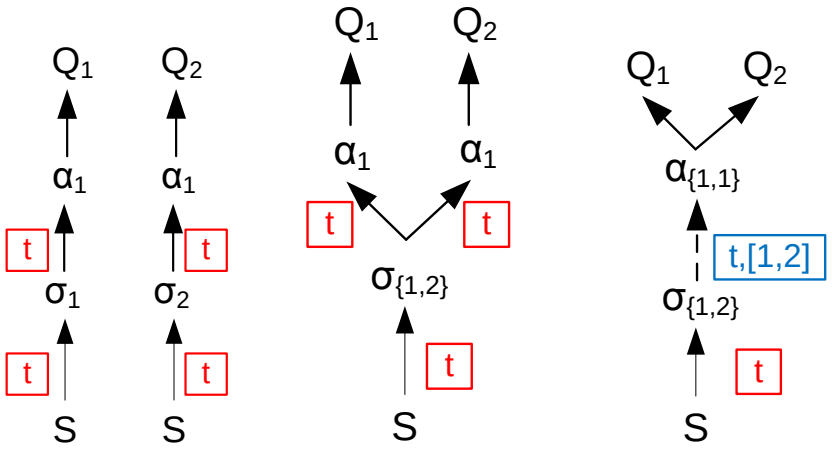
Example 21 *Figure 5.1(a) shows two given logical queries Q_1 and Q_2, where σ_1 and σ_2 are selection operators, and α_1 denotes a sliding window aggregation operator, occurring in both queries. Note that we use the query name to denote its output stream name.*

Let $\sigma_{\{1,2\}}$ denote the m-op implementing σ_1 and σ_2 with predicate indexing. It produces two output streams, respectively corresponding to the output streams of σ_1 and σ_2 in Figure 5.1(a). Figure 5.1(b) shows the query plan using $\sigma_{\{1,2\}}$.

Suppose tuple t in stream S satisfies both σ_1 and σ_2. In Figure 5.1(a), an output tuple is produced by both σ_1 and σ_2. In Figure 5.1(b), an output tuple is produced by $\sigma_{\{1,2\}}$ on each of its two output streams.

5.2.3 Transformation Rules on m-ops

We now extend the traditional transformation rules, which operate on query plans composed of physical operators, to *multi-query transformation rules*, or *m-rules* for short. M-rules operate on query plans composed of m-ops. Similar to a traditional transformation rule, an m-rule consists of a pair of *condition* and *action* functions [70]. The condition function is a boolean side-effect-free function on the input query plan to identify opportunities for sharing. Once a sharing opportunity is identified among a set of operators in the query plan, the action function modifies the query plan by replacing that



(a) Input Queries (b) Plan with Selection m-op (c) Plan with Channel

Figure 5.1: Query Plans in RUMOR (Red Rectangles Represent Stream Tuples; the Blue Rectangle is a Channel Tuple)

set of operators with a single m-op. In this case, we say the m-rule *maps* these operators to the same m-op, or it *merges* these operators.

More precisely, the condition of an m-rule is a function from the powerset of the set of all possible m-ops to $\{\text{true}, \text{false}\}$. For a given set of m-ops, the rule can only be applied if the function evaluates to true.

The action of an m-rule is another function. This function maps the set of input m-ops (for which the condition function evaluated to true) to a single m-op, referred to as the *target m-op*, which implements the input m-ops more efficiently with an MQO technique. In the query plan, we simply replace all edges that previously connected other operators with the to-be merged input operators by edges to the corresponding input and output streams of the target m-op.

For example, we can model predicate indexing for equality predicates on a single attribute as an m-rule as follows. The condition of the m-rule evaluates to true only for

a set of selection operators that all read the same stream and whose selection predicate is an equality predicate on the same attribute A. The action rule then replaces them with a target m-op that uses a hash index on attribute A for a more efficient evaluation of the selection predicates of these operators.

5.2.4 Expressing MQO Techniques with m-ops and m-rules

Most of the previously proposed specialized MQO techniques share work among multiple operators reading the *same stream(s)*. Since RUMOR, like a relational engine, uses query plans as the processing model, we focus on the MQO techniques for the three key relational operator types: selection, join and aggregation (see the first three rows in Table 5.2.4). Notice that in data stream processing systems, join and aggregation operators usually contain window specifications to prevent unbounded memory consumption. Also, aggregation operators may contain optional group-by specifications. For each operator type τ, we name the corresponding m-rule s_τ, indicating that it is an m-rule for instances of operator τ that all process the same input stream(s). The remaining rows in Table 5.2.4 will be described later in this chapter. The set of m-rules presented in Table 5.2.4 is not intended to be complete—the extensible nature of rule-based query optimization allows for adding new rules.

Note that given a set of operators, even in the absence of any nontrivial MQO technique to share their work, it may still be beneficial to have an m-rule to map them to a single m-op. As a concrete example, suppose we have a set of sliding window aggregation operators reading the same stream. These operators may have different aggregate functions and group-by specifications. If we map them to a single m-op, we can potentially a) share the sliding window content in the operator states, b) share the window

Table 5.1: Representative m-rules to Express Existing and New MQO Techniques

m-rule name	Set of input operators to which the m-rule is applicable	Target m-op
s_σ	A set of selection operators which read the same stream	Predicate indexing [33, 63]
s_α	A set of aggregation operators which read the same stream, with the same aggregate function but potentially different group-by specifications	Shared aggregate evaluation [99]
$s_{\triangleright\triangleleft}$	A set of join operators which read the same two streams, with the same join predicate but potentially different window lengths	Shared join evaluation [43]
c_α	A set of aggregation operators reading sharable streams, with the same definition	Shared fragment aggregation [55]
$c_{\triangleright\triangleleft}$	A set of join operators which read sharable streams, with the same definition	Precision sharing join [54]
$s_;$ (or s_μ)	A set of ; (or μ) operators reading the same two streams, with the same definition	Common Subexpression Elimination (Section 5.4.3)
$c_;$ (or c_μ)	A set of ; (or μ) operators which a) have the same definition b) read sharable input streams for the first input stream parameter, where these input streams are produced by the same m-op c) read the sme input stream for the second input stream parameter	Channel Based MQO (Section 5.4.4)

computation if some aggregate functions are related, such as deriving average from sum and count, and c) exploit memory locality by evaluating the aggregate operators consecutively on each incoming stream tuple. For this reason, we add the following *default m-rules* to RUMOR: For a set of operators of the same type reading the same input stream(s), if no other m-rule is applicable, we map them to an m-op, which implements them by sequentially evaluating them on each input tuple.

5.3 RUMOR: Part II

In Section 5.2, we have shown how to use the two abstractions, m-op and m-rule, to express a set of existing MQO techniques, including predicate indexing [33, 63], multiple aggregate processing with different group-by specifications [99], and shared join evaluation [43]. All of these techniques attempt to share work among multiple operators reading the *same* stream(s).

However, there are other MQO techniques which are able to share work among operators reading *different* input streams, provided that these different streams may have tuples of the same content. Multiple streams often share tuples of the same content, when they are produced by operators of a particular type, which read the same input stream. A simple example are the output streams of different selection operators on the same input stream. An input tuple might satisfy the selection predicate of several of these operators.

These MQO techniques include precision sharing join [54] and shared fragment aggregation [55]. To continue Example 21 from Section 5.2.2, the shared fragment aggregation technique can be applied to the two aggregate operators in Q_1 and Q_2 to share work, since they have the same definition. As another example, consider n projection operators with the same projection specification, but with different input streams S_1 through S_n. Suppose each of the n input streams contains a tuple of the same content. In this case, if we can manage to represent these n tuples of the same content together as a single tuple, we can perform projection only once on that tuple, and produce a single output tuple recording the fact that it belongs to the output streams of the n projection operators. This is significantly more efficient than performing n projections and creating n output tuples.

In this section, we propose an abstraction to model such MQO techniques in RUMOR. We refer to it as a *channel*. Channels generalize and replace streams in RUMOR (Section 5.3.1). We then describe how to decide which streams should be replaced with channels in the query plans in RUMOR (Section 5.3.2). Finally, we add a new set of m-rules and m-ops leveraging channels, which express both existing and new MQO techniques (Section 5.3.3).

5.3.1 Extending Streams to Channels

Logically, a channel is equivalent to the union of a set of streams with "compatible schemas", the same requirement that the input streams of a union operator need to satisfy. Different from stream union, a channel keeps track of which original stream a tuple belongs to. We say the channel *encodes* its set of streams.

More formally, a channel encodes a set of data streams with union-compatible schemas as follows. The channel is defined as the union of its streams, but each stream tuple has an additional attribute—called a *membership component*. The membership component specifies the set of streams to which this tuple belongs. For efficiency, the membership component is usually implemented by a bit vector.

Through the use of a channel we can share work in two ways. First, when identical tuples from different streams are encoded as a single channel tuple, their *space representations* are shared. Second, when multiple streams are encoded into the same channel, the *computation* of their consumer operators may be shared.

Clearly, channels generalize streams. In RUMOR, they take the place of streams as the input and output of m-ops. For each m-op, the input (resp. output) channels together

partition the set of input (resp. output) streams of this m-op. When an m-op o processes an input channel tuple t, a decoding and an encoding step are involved as follows. o first determines to which set of input streams t belongs, so that o conceptually only evaluates those physical operators implemented by o that take this tuple as input. This is the decoding step. Similarly, when o is about to produce a set of output tuples for input channel tuple t, it needs to encode them into a set of channel tuples, and then write them to the appropriate output channels. This is the encoding step.

Note that the decoding and encoding steps can often be implemented very efficiently, or might actually not be necessary at all. For example, consider an m-op $\pi_{\{1,\cdots,n\}}$ implementing n projections with the same projection specification, but with different input streams S_1 through S_n. Suppose these n input streams are encoded by channel C, and the n output streams are encoded by channel D. In this case, for each input channel tuple t from C, $\pi_{\{1,\cdots,n\}}$ only needs to perform projection once and to produce one output channel tuple in D, while keeping the membership component of t intact in the output D tuple.

To continue Example 21, we can use a channel to encode the two output streams of $\sigma_{\{1,2\}}$ in Figure 5.1(b), resulting in the query plan shown in Figure 5.1(c). Here the dashed arrow represents the channel, and $\alpha_{\{1,1\}}$ represents the aggregation m-op, implemented by the shared fragment aggregation technique described in [55]. Suppose an input tuple t from stream S satisfies both predicates in $\sigma_{\{1,2\}}$. $\sigma_{\{1,2\}}$ then produces an output channel tuple, represented by the blue rectangle in Figure 5.1(c). That channel tuple has the same content as the input tuple t, but is associated with a membership component denoted as [1,2], indicating that it belongs to both output streams of $\sigma_{\{1,2\}}$.

Note that ideas similar to channels were used for specific MQO algorithms for joins and aggregates in relational engines [54, 55]. We propose the concept of a channel as a

general mechanism for sharing work among a wider spectrum of operators beyond join and aggregation. For example, we showed above how channels can be used to share work among a set of projection operators. Also, in Section 5.4, we will show how it can be used for our newly proposed MQO techniques for event processing queries.

5.3.2 Mapping Streams to Channels

Channels are a powerful mechanism that allows us to aggressively share work among operators that read even different streams. Given that a set of k streams can be mapped to 1 through k channels, how do we decide the mapping? There are clear trade-offs. If two streams S_i and S_j are encoded into one channel, then stream tuples of the same content can share storage by being represented as the same channel tuple. Also, if the consumer operators of S_i and S_j have the same definition, the evaluation on channel tuples will be more efficient than evaluating tuples from stream S_i and S_j separately. The disadvantage of mapping multiple streams to the same channel is the additional runtime overhead. Time-wise, with multiple streams being mapped to the same channel, the consumer m-op of this channel now has to process the membership component of each input tuple. Space-wise, each channel tuple has to carry the membership component.

One way to decide which streams to encode into a single channel is by using a detailed cost model that reflects the runtime and space tradeoffs of using channels. Examining this in detail is beyond the scope of this chapter and hence left for future work. Here we present a lightweight algorithm that, based on our experience, performs very well in practice and adds very little overhead for encoding and decoding tuple membership components.

Our proposed algorithm for deciding which streams to merge into a single channel is

based on the concept of *sharable* streams. Two streams S_1 and S_2 are sharable, denoted $S_1 \sim S_2$, if the following holds:

Base case 1. If $S_1 = S_2$, then $S_1 \sim S_2$.

Base case 2. If S_1 and S_2 are produced by two stream sources that are labeled to be sharable, then $S_1 \sim S_2$.

Output of unary ops. For any unary operators o_1, o_2, if $S_1 := o_1(T_1), S_2 := o_2(T_2), o_1 = o_2,$ and $T_1 \sim T_2$, then $S_1 \sim S_2$.

Output of binary ops. For any binary operators o_1, o_2, if $S_1 := o_1(T_1, U_1), S_2 := o_2(T_2, U_2), o_1 = o_2, T_1 \sim T_2,$ and $U_1 \sim U_2$, then $S_1 \sim S_2$.

Special case for selection. For a selection operator that reads T and produces S, $S \sim T$.

Symmetry $\forall S_1, S_2 : S_1 \sim S_2 \Rightarrow S_2 \sim S_1$

Transitivity $\forall S_1, S_2, S_3 : (S_1 \sim S_2 \wedge S_2 \sim S_3) \Rightarrow S_1 \sim S_3$

Clearly, $\sim$ is an equivalence relation and it generalizes the stream identity relation $=$. This makes $\sim$ very efficient to compute and store.

There are two additional criteria to be considered before encoding streams in the same channel. First, even if two streams are sharable as defined above, if they are produced by two different m-ops, we still cannot benefit from encoding them with the same channel. This is because the evaluation of these two m-ops in the query engine is not "coordinated". Concretely, even if these two m-ops produce tuples of the same content, additional efforts have to be taken to verify that these output tuples are indeed of the same content, and if so, to replace them with a single channel tuple.

Second, if the two sharable streams are consumed by m-ops that cannot share any work, there is no benefit in encoding them with the same channel either. Typically two m-ops reading different streams can share work only if they have exactly the same definition. For example, two selection operators with the same predicate, two projection operators with the same projection specification, or two aggregation operators with the same aggregate function and group-by specification can share work when reading two different input streams that are sharable. We only consider this type of work sharing with channels in this chapter.

To conclude, given a set of streams S_1 through S_n, we map them to the same channel, only if a) the S_i's belong to the same equivalence class defined by $\sim$, b) the S_i's are produced by the same m-op, and c) the consumers of the S_i's have the same definition. These criteria, referred to as *channel-based MQO sharing criteria*, are used in RUMOR. When these criteria are satisfied, we map the consumers of the S_i's to the same m-op, achieving effective work sharing among them.

The above sharing criteria may appear restrictive. However, they are often satisfied by sets of operators in a workload with many queries, since there is usually much commonality among different queries. For example, precision sharing join [54] and shared fragment aggregation [55] are both implicitly based on the above criteria, additionally limited to join operators and aggregation operators, respectively.

5.3.3 Expressing MQO Techniques with Channels

To benefit from channels, we add the following new m-rules. For each operator type τ (e.g. selection, join, aggregation), we add an m-rule which identifies operators of type τ whose input streams satisfy the channel-based MQO sharing criteria defined at the end

of Section 5.3.2. It then maps these operators to a single m-op. We refer to this m-rule as c_τ, indicating that this is an m-rule for operators τ processing tuples from the same channel. For example, the fourth and fifth m-rule in Table 5.2.4 respectively express shared fragment aggregation [55] and precision sharing join [54].

Note the interesting duality between the two m-rules s_τ (defined in Section 5.2.4) and c_τ of an operator type τ. s_τ is applicable to a set of *sharable* operators (i.e., operators of type τ) reading the *same* stream(s), whereas s_τ is applicable to a set of operators of the *same* definition, reading *sharable* stream(s). Assuming τ is unary, we present a pictorial illustration for the difference between s_τ and c_τ. In Figure 5.2, the enclosing rectangle denotes the set of unary operators of type τ, reading sharable streams. Each row labeled S_i corresponds to a subset of operators of type τ, reading the same stream S_i. Each application of s_τ will pick a row of operators here, and map them to an m-op. Repeated applications of this m-rule therefore form a partition of this set of operators. The setting of Figure 5.3 is similar to that of Figure 5.2. Each column corresponds to a set of operators of type τ with the same definition, reading a set of sharable input streams $S_1, S_2, \cdots$. One application of c_τ selects a column of operators, and maps them to an m-op. Repeated applications of c_τ therefore also form a partition of this set of operators.

As a result, for any operator in the shaded region X; i.e., an operator with definition o_1, reading stream S_1, both s_τ and c_τ are applicable to it. Similar to many other rule-based applications, different orderings of m-rule applications therefore may result in different optimized query plans. It may be useful to assign priorities to the m-rules, or to have a cost model in deciding the ordering of m-rule applications. These extensions are however beyond the scope of this chapter.

To summarize, Table 5.3.3 shows the newly proposed abstractions in RUMOR, and

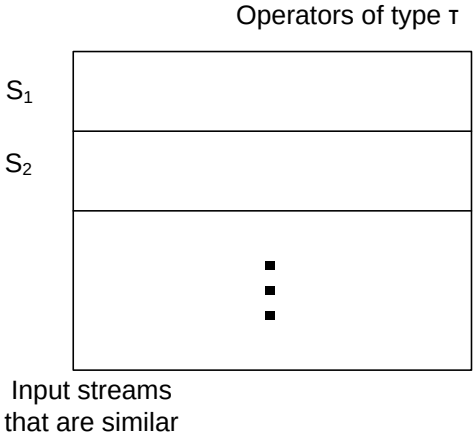


Figure 5.2: $\mathcal{R}_1$ Applied to a Set of Operators of Type τ

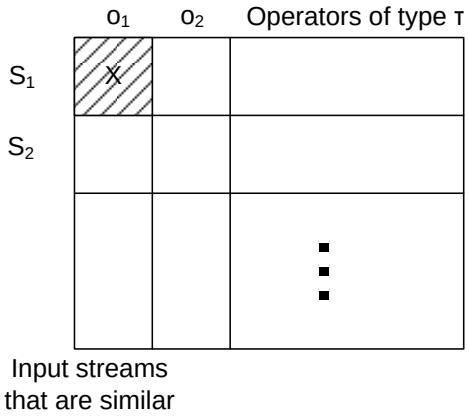


Figure 5.3: $\mathcal{R}_2$ Applied to a Set of Operators of Type τ

Table 5.2: Correspondence between new and existing abstractions for building a stream system

Existing abstraction	**RUMOR abstraction**
physical operator	m-op (Section 5.2.2)
transformation rule	m-rule (Section 5.2.3)
stream	channel (Section 5.3)

their correspondences with existing abstractions.

5.4 Integrating MQO Techniques for Event Engines

In Section 5.2 and Section 5.3, we have presented RUMOR, as well as how the MQO techniques for REs can be integrated into RUMOR. In this section, we describe how the MQO techniques for EEs can be integrated. This is a more challenging task, as EEs are often based on automata, instead of query plans composed of relational operators. On

the other hand, if we are able to integrate the MQO techniques for both REs and EEs into RUMOR, we will be able to build an expressive and scalable stream system unifying REs and EEs, of which there are obvious and significant benefits (Section 5.4.1). Our solution consists of two parts. First, we translate automata into query plans (Section 5.4.2), second, we express the MQO techniques designed for automata in RUMOR (Section 5.4.3). Interestingly, new channel-based MQO techniques for EEs, which have not been proposed before, can enable further computation and state sharing among multiple queries (Section 5.4.4).

5.4.1 A Motivating Scenario for Unifying REs and EEs

The separation of stream processing systems into REs and EEs prevents us from efficiently supporting a large class of stream applications, whose query workloads demand functionality from both types of stream systems. In the remainder of this subsection, we provide a motivating example for the unification, which we will revisit in the following subsections to supply the query formulations and opportunities for MQO.

In performance monitoring of computer systems [81, 32], each stream corresponds to readings of a particular performance counter, such as the amount of current CPU consumption of a particular thread or process. Multiple users can register continuous queries in a stream system; e.g. to compute the average CPU load in a time-based sliding window, or to raise alerts on specified conditions and optionally to perform certain actions, such as terminating resource hogging processes.

Input streams. We assume the following input stream schema: `CPU(pid, load; ts)`, indicating the CPU load of each process in the system. `pid` denotes process ID; `load` denotes CPU load; `ts` denotes the required timestamp attribute for

each stream.[1]

Queries. We identify two interesting characteristics that query workloads exhibit, which pose challenges in RSE.

First, there may exist a large number of concurrent queries in the system, since different queries may be registered to monitor the behavior of different processes. Furthermore, for a particular process, different monitoring conditions may be posed in different queries. To obtain high throughput, it is crucial to apply MQO techniques to these queries. In this chapter, we show how to use our rule-based MQO framework to express and integrate existing and new MQO techniques.

Second, some performance monitoring queries demand functionality from both CQL-style queries supported by an RSE, and pattern matching queries supported by an event engine. We refer to such queries as *hybrid queries*.

Consider the following hybrid query, which detects processes that are ramping up in CPU consumption. This query combines the functionality of sliding window aggregates (supported by an RSE) for smoothing the incoming performance counter readings, and the functionality of event pattern detection (supported by an event engine) for finding a monotonically increasing sequence in CPU load consumption.

Query 1 *For a particular process p, smooth the CPU load value, by replacing the current CPU load for p with an average load of p over the last 5 seconds. Call the smoothed stream* `SMOOTHED`*. Next, find in* `SMOOTHED` *an event pattern composed of a sequence of monotonically increasing CPU loads on p, where this sequence pattern satisfies a customizable starting condition θ_s, and a fixed stopping condition, say $CPU.load > 90$.*

[1] In practice there are more performance counters than just CPU, such as for memory and disk. The streams' schemas also involve more attributes. We simplify the scenario here for ease of presentation, and refer the interested readers to [32] for a detailed view of the Windows NT performance counters.

An example starting condition is $\theta_s = CPU.load < 20$.

Such a hybrid query is not supported by a typical RE or EE today. Although it is possible to extend an existing system to support a few instances of hybrid queries, it becomes challenging to efficiently support a large number of hybrid queries as in Query 2.

Query 2 *We have a set of queries* $\{Q_1, \cdots, Q_n\}$*, where each* Q_i *differs from Query 1 only in the starting condition* θ_s*.*

For this query workload, it is possible to *manually* construct query plans that achieve good computation sharing. However, the focus of this chapter is to automate MQO with RUMOR. In the remainder of Section 5.4, we will revisit this query workload and describe how automated MQO is achieved in RUMOR. This is however predicated on the understanding of how the MQO techniques for EEs are integrated into RUMOR, which we present below.

5.4.2 Translating Automata to Query Plans

Event Engines are often based on automata [22, 36, 29, 95]. In order to integrate the MQO techniques for EEs into RUMOR, our first step is to model the automata used in EEs as query plans in RUMOR. In the following text, we choose Cayuga, a representative EE, and show how to express the Cayuga automata queries as query plans in RUMOR. It is possible to integrate other event engines, such as SASE [95], within the RUMOR framework.

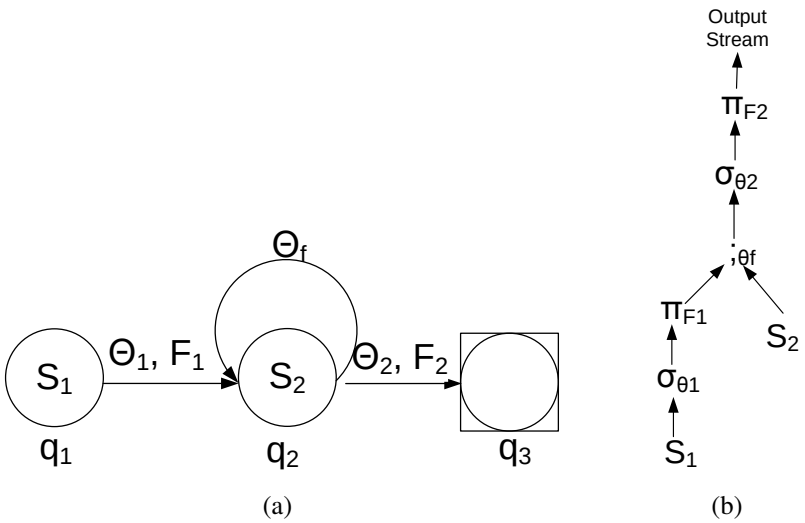


Figure 5.4: (a) An Example Cayuga Automaton, and (b) the Equivalent Query Plan

Cayuga event pattern queries are compiled into variants of automata, to be evaluated in the query engine. First we briefly review this automata model, and refer to [29, 30] for the formal processing model accompanied with examples, as well as the implementation. In an automaton, all the out-going edges of a particular state read the same input stream. For this reason, input streams can also be associated with states. Each state has a fixed schema. A forward edge connects two states, whereas a filter or rebind edge is a self-loop edge on a state. When an automaton instance traverses a filter edge, its content is not modified. This is not the case when an automaton instance traverses a forward or rebind edge. Each filter edge is associated with a predicate. Each forward or rebind edge is associated with a predicate, and a schema map function. A schema map function can rename and project attributes, as well as introducing new attributes via simple arithmetic computation or user-defined functions. It is similar to a SQL projection operator (which implements the SQL SELECT clause).

For example, a Cayuga automaton is shown in Figure 5.4(a). q_1, q_2 and q_3 are the names assigned to the states. The input stream of each state is shown within the state.

The edges connecting q_1 to q_2, and q_2 to q_3 are forward edges. The pair θ_i, F_i above each forward edge denotes its predicate and schema map function. The self-loop edge on q_2 is a filter edge. θ_f above the filter edge denotes the filter predicate. If q_2 also had a rebind edge, it would have been drawn as another self-loop edge below q_2.

To express a Cayuga automaton as a query plan in RUMOR, we need to introduce two new m-ops into RUMOR. Given an automaton state with a filter edge but no rebind edge, that state will be translated into an m-op denoted as `;`, whose semantics is the same as its counterpart in Cayuga algebra [29]. Intuitively, `;` is a sequence operator concatenating two input events. Similarly, an automaton state with a rebind edge is translated into an m-op denoted as μ, whose semantics is also the same as its counterpart in Cayuga algebra. μ is an iterative version of `;`, capable of concatenating an unbounded number of input events into an event sequence pattern. The formal definitions of `;` and μ can be found in [29].

We now illustrate through an example how to translate a Cayuga automaton into an RUMOR query plan. For the Cayuga automaton in Figure 5.4(a), we start with the input stream S_1, read by q_1, the start state of the automaton. The predicate θ_1 on the forward edge of q_1 is translated to σ_{θ_1} in the query plan. Similarly, the schema map function F_1 on the same edge is translated to π_{F_1} in the query plan, reading the output stream of σ_{θ_1}.[2] Next, we translate state q_2 into a binary operator $;_{\theta_f}$, reading the output stream of π_{F_1} as well as S_2. Finally, the forward edge from q_2 to q_3 is translated in a similar way as the forward edge from q_1 to q_2. We use σ_{θ_2} and π_{F_2} respectively to implement the predicate θ_2 and the schema map function F_2 on that forward edge. The output stream of π_{F_2} is equivalent to the output stream of the automaton. This finishes the translation. The resulting query plan is shown in Figure 5.4(b).

[2]Here π denotes the more expressive SQL projection operator, as opposed to the projection operator in relational algebra.

The translation of a Cayuga automaton involving states with rebind edges is similar. For example, if state q_2 in Figure 5.4(a) also has a rebind edge with predicate θ_r, then the operator $;_{\theta_f}$ in Figure 5.4(b) will be replaced with μ_{θ_f,θ_r}.

We omit the formal specification of the automaton-to-query plan translation.

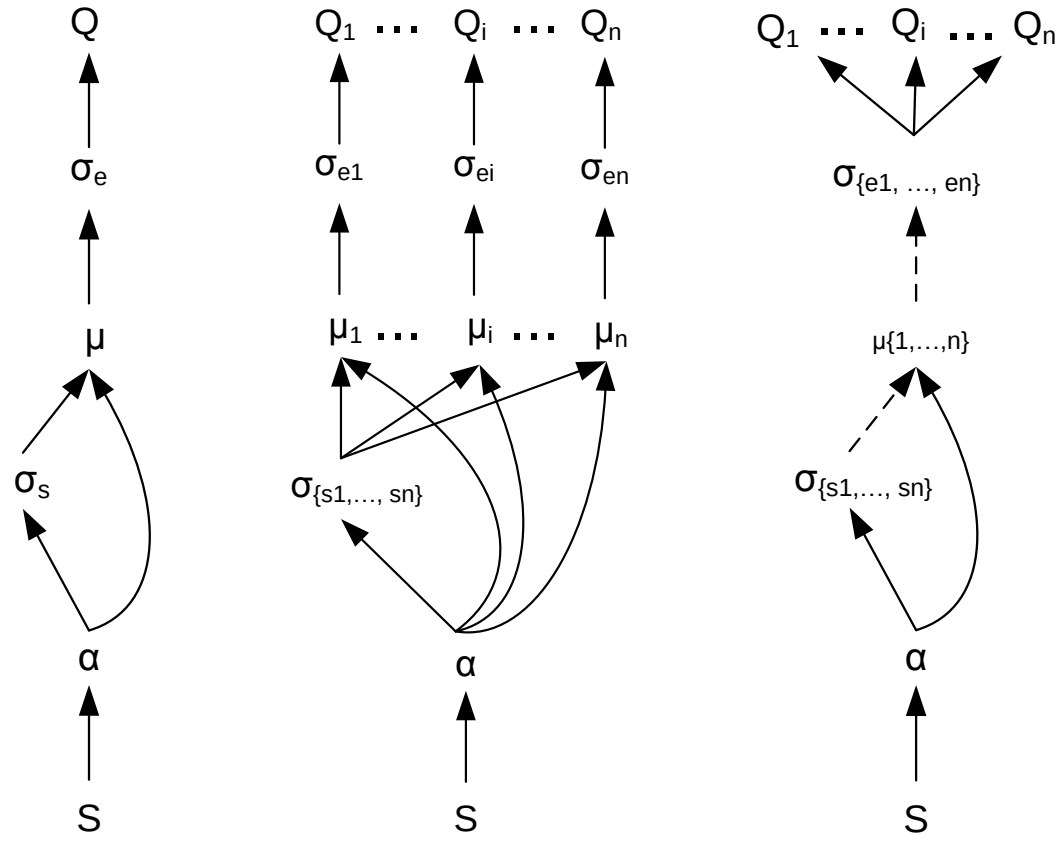
Example 22 *The RUMOR query plan for Query 1 in Section 5.4.1 is shown in Figure 5.5(a). For clarity, we omit projection operators and the parameters of some operators in the query plan.*

The input stream is denoted as S. α denotes the sliding window aggregator operator for smoothing the CPU load readings of each process. σ_s and σ_e are respectively the starting and stopping conditions. μ builds up the event sequence pattern consisting of monotonically increasing values in the CPU loads of a particular process. Finally, as in Example 21, we use the query name Q to denote its output stream name.

5.4.3 Expressing Automata Based MQO Techniques in RUMOR

In Section 5.4.2, we achieved the unification of an RE and an EE on a single automaton level, by translating an automaton into a query plan. However, to efficiently process a large number of event pattern queries, an EE often employs a set of MQO techniques specially designed for automata. It is not clear how we can evaluate the event pattern queries in the form of query plans at the same level of efficiency as an EE, unless we can express the MQO techniques for EEs in RUMOR to share work among the query plans.

In this subsection, we again use Cayuga as a representative EE which adopts MQO, and express *all* its MQO techniques by m-rules and m-ops in RUMOR. To our knowledge, no other MQO techniques for EEs have been proposed outside the context of



(a) Query Plan for Query 1

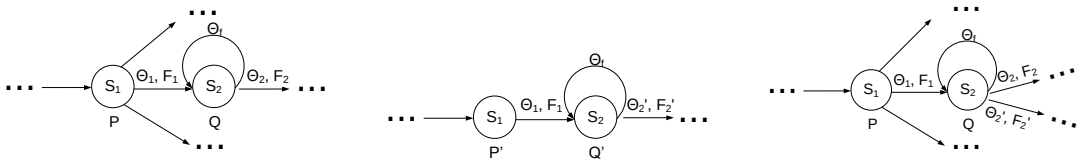
(b) Query Plan for n Instances of Query 2 Without Channel

(c) Query Plan for n Instances of Query 2 With Channel

Figure 5.5: RUMOR Query Plans for the Motivating Queries in Section 5.4.1 (Omitting Projection Operators for Clarity)

Cayuga, but we will show in Section 5.4.4, that when new MQO techniques for EEs are available, how they are integrated into RUMOR.

Given that we have introduced two new operators **;** and μ into RUMOR in Section 5.4.2, next we add a new m-rule for each of these two operators. The m-rule $s_;$ for **;** (resp. s_μ for μ) maps a set of **;** operators (resp. μ operators) to an m-op, if they read the same pairs of streams, and have the same definition. These two m-rules are shown in the second-to-last row in Table 5.2.4.



(a) Current NFA Forest before Merge

(b) The Query Automaton to Merge

(c) Current NFA Forest after Merge

Figure 5.6: Cayuga Automata State Merging Process

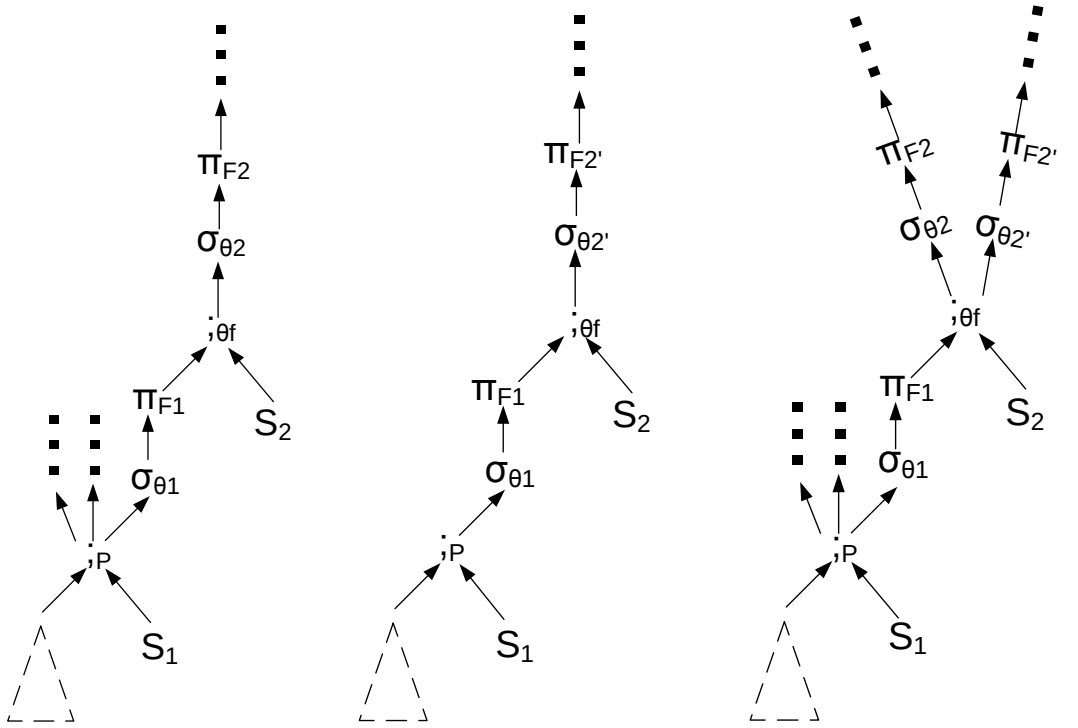
There are two major categories of MQO techniques in Cayuga, state merging and indexing. We show how these techniques can be expressed by the m-rules.

State merging. The first type of state merging in Cayuga is prefix state merging. Intuitively, given an existing automaton $\mathcal{F}$, and a new input automaton A, A can be merged into $\mathcal{F}$ by identifying the longest prefixes of $\mathcal{F}$ and A that are identical, and share the two prefixes in the merged automaton.

As a concrete example, the existing automaton and the input automaton to merge are shown respectively in Figure 5.6(a) and 5.6(b). In this example, suppose inductively that state P and P' have been merged, and state Q and Q' read the same stream (in this case it is S_2), then we can merge states Q and Q'. The resulting automaton is shown in Figure 5.6(c).

This prefix state merging technique is expressed by the m-rules $s_;$ and s_μ together. We illustrate this with the above example. The query plans corresponding to the existing automaton and the input automaton are shown respectively in Figure 5.7(a) and 5.7(b). Note that the operator $;_{\theta_f}$ in Figure 5.7(a) and in Figure 5.7(b) respectively implement state Q and Q' in the corresponding automata.

Suppose inductively that the common sub-expressions below operator $;_{\theta_f}$ in Figure 5.7(a) and in Figure 5.7(b) have been merged. The m-rule $s_;$ corresponding to $;$ is now



(a) Current Query Plan before Merge

(b) The Input Query Plan to Merge

(c) Current Query Plan after Merge

Figure 5.7: RUMOR Query Plans for Cayuga Automata

applicable to $;_{\theta_f}$ in both figures, since by assumption they read the same pair of streams, and have the same definition. Hence, they are mapped by the m-rule $s_;$ to the same m-op, which we still denote as $;_{\theta_f}$, since it has the same definition as the two input operators that are merged. The resulting query plan is shown in Figure 5.7(c). The prefix state merging performed on multiple μ operators can be done in a similar way.

Note that we have translated the prefix state merging MQO technique on automata into the well-known MQO technique on query plans: Common Subexpression Elimination (CSE).

Another type of state merging in Cayuga is to inline one automaton into another automaton that reads the output of the former. For example, a Cayuga query $(S_1 ; S_2) ; S_3$

can be naively implemented by two automata as follows. The first automaton A implements S_1 ; S_2, and produces an intermediate stream S'. The second automaton B implements S' ; S_3. In this case, we can inline A into B, by replacing the forward edge of the start state of B with A, eliminating the need for producing the intermediate stream S'. This is supported in RUMOR as query plans are composable.

In addition, in RUMOR, we have more opportunities for inlining, illustrated as follows. For an input Cayuga query that is not left-associative, such as S_1 ;(S_2 ; S_3), it has to be implemented by two Cayuga automata A and B, where A implements S_2 ; S_3, producing an intermediate stream S', and B implements S_1 ; S'. This is referred to as resubscription in [29], and in this case A cannot be inlined into B. However, this query can be implemented by a *single* query plan, which effectively inlines the query plan corresponding to A to that corresponding to B.

Automaton indexing. There are three types of indices in Cayuga. Below we describe how to express the FR Index technique in RUMOR. The other two indices, Active Node Index and Active Instance Index [29, 30], are handled similarly.

Forward-Rebind Index (FR Index) is a per-state index on some of the predicates of forward/rebind edges of its associated state. For example, in Figure 5.6(c), the predicates θ_2 and θ_2' associated with the forward edges going out of state Q can be managed by an FR Index. For an incoming event e from stream S_2 which satisfies the filter edge predicate θ_f associated with state Q, e will be used to probe this FR index to obtain the set of satisfied predicates associated with the forward edges (i.e., a subset of $\{\theta_2, \theta_2'\}$).

An FR index on state q can be expressed by the m-rule s_σ in RUMOR as follows. Let the operator corresponding to q in the translated RUMOR query plan be o (o is of type ; or μ). For those selection operators that are consumers of the output stream of o, we

apply the m-rule s_σ to map them to the same m-op. That m-op effectively implements the FR index for the translated query plan.

For example, recall that for the automata shown in Figure 5.6(c), its corresponding query plan in RUMOR is shown in Figure 5.7(c). For the FR Index on the forward edges of state Q in Figure 5.6(c), which we described above, it can be expressed by applying s_σ to σ_{θ_2} and $\sigma_{\theta_2'}$ above the $;_{\theta_f}$ operator in Figure 5.7(c).

To conclude, we have shown that with RUMOR, all the MQO techniques employed by Cayuga can be expressed as m-rules on RUMOR query plans. Hence, asymptotically, the evaluation efficiency of a set of event pattern queries in RUMOR is at least as good as that in the Cayuga engine, as is confirmed by our experiments.

Example 23 *The RUMOR query plan for n query instances of Query 2 in Section 5.4.1, denoted as Q_1 thorugh Q_n, is shown in Figure 5.5(b).*

The aggregation operator α is shared by all n queries. It produces a single stream called `SMOOTHED` *in Query 1, and multiplexes it to all its consumer operators.*

The n starting conditions are implemented by the m-op $\sigma_{\{s_1,\cdots,s_n\}}$, which produces n output streams corresponding to that of σ_{s_1} through σ_{s_n}.

μ_i builds the event sequence pattern for query Q_i. It reads the two streams produced by σ_{s_i} and α respectively. Its output stream is consumed by σ_{e_i}, the stopping condition for Q_i.

Note that in Example 23, even though the μ_i operators have the same definition, they cannot share work, since their first input streams are different. The same observation holds for the σ_{e_i} operators. This is a limitation for RUMOR without channels, and is

also the case for Cayuga automata. We will show in the next subsection how to use channels to overcome this limitation.

5.4.4 Query Plans with Channels

In the previous subsection, we have shown how to express all of the Cayuga MQO techniques as m-rules and m-ops in RUMOR. In this subsection, we show that, somewhat surprisingly, there are event pattern queries that can be evaluated more efficiently in the form of RUMOR query plans than in Cayuga engine. This is due to the rise of new MQO opportunities with channels, illustrated through the following example.

Example 24 *Let us revisit Query 2 in Section 5.4.1, and consider how to process n instances of this query more efficiently than the query plan shown in Figure 5.5(b). The sliding window aggregation part of these queries for smoothing the input stream is already shared. For the pattern matching part, a good evaluation strategy is to first evaluate the starting conditions in the n queries. If any subset of them is satisfied, we remember this information and continue to match the monotonic sequence patterns of these queries, implemented by the μ operators. When the stopping condition is satisfied, we then use the information we remembered for which θ_{s_i}'s are satisfied to produce result tuples for the right set of queries.*

The RUMOR query plan implementing this evaluation strategy is shown in Figure 5.5(c). As in Figure 5.1(c), we use dashed arrows to represent channels. However, note that this evaluation strategy is outside of the Cayuga automata model, and therefore cannot be used by the Cayuga engine.

In order to produce the desired query plan with channels shown in Figure 5.5(c), we add the follow one m-rule for ; and μ each. The m-rule for ;, denoted as $c_;$, maps a set of ; operators to a single m-op, if these operators satisfy a) they have the same definition b) they read sharable input streams for the first input stream parameter, where these input streams are produced by the same m-op c) they read the same input stream for the second input stream parameter. In this case, we encode the first input streams of these operators with a channel. The new m-rule for μ works in a similar way. These two m-rules are shown in the last row in Table 5.2.4. The stream sharability computation and the channel-based MQO sharing criteria defined in Section 5.3 are extended accordingly for ; and μ.

We now show how to use the m-rules to optimize n instances of Query 2, denoted as Q_1 through Q_n. Starting from the query plan in Figure 5.5(b), we first apply the m-rule s_σ to the set of starting conditions in $Q_1, \cdots, Q_n$, and encode their output streams with a channel C. Next, we apply the m-rule c_μ to the set of μ operators in the n queries, and again encode their output streams with a channel D. Finally, we apply the m-rule c_σ to the set of stopping conditions in $Q_1, \cdots, Q_n$, resulting in a selection m-op that reads channel D as input, and produces n output streams for the n queries.

5.5 Performance Evaluation

We have implemented in Java a prototype stream engine based on RUMOR, which is capable of processing RE queries, EE queries, as well as hybrid queries. In this section we report the performance of our engine in evaluating the optimized query plans. The experiments are conducted on a machine with Intel Pentium D 2.80 GHz processor and 2 GB main memory, running Sun Java Hotspot Server VM 1.6.02 on Windows Vista.

To leverage the JVM just-in-time code optimization, for each experiment, we first process the input stream for a few iterations, before we start to measure throughputs. To reduce experimental variance, we perform each experiment for ten times, and report the average throughputs we measured.

5.5.1 Setup

We first use a synthetic benchmark to measure the performance of our system for processing event pattern queries and hybrid queries. We do not measure the performance of RE queries, since our query-plan-based approach to evaluating RE queries is similar to previous work, where the performance of query plans has been well studied [11, 24].

The stream schema we choose consists of 10 integer attributes, denoted as $a[0], \cdots, a[9]$, and 1 (integer) timestamp attribute. We generate two streams conforming to this schema, denoted as S and T, as follows. The generated stream tuples have consecutive timestamps, starting from 0. For each tuple, we set its 10 integer attributes to integer values from 0 to 999 chosen uniformly at random. We interleave the generation of tuples for S and T. That is, tuples with timestamps 0, 2, $\cdots$ belong to S, and tuples with timestamps 1, 3, $\cdots$ belong to T. For each experiment, we generate a total of at least 100000 tuples, and feed the tuples from S and T in their timestamp ordering.

We use the following common parameters to generate query loads. For each randomly generated query, we choose a window length for it from 1 to 1000, where 1000 is the default domain size for generating window lengths. Each window length is chosen with a Zipfian distribution, favoring larger windows (i.e., a window of length 1000 is most likely to be chosen). The default Zipfian parameter value is 1.5. The Zipfian distribution is to model commonality among queries that is often observed in real, large-scale

Table 5.3: Parameters (default values)

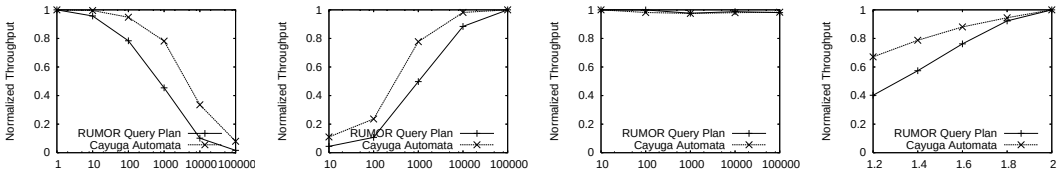
Variable	Default Value
Number of queries	1000
Number of attributes in stream schemas	10
Constant domain size	1000
Window length domain size	1000
Zipfian parameter	1.5

workloads. The parameters are summarized in Table 5.5.1.

5.5.2 Event Pattern Queries

In Section 5.4, we have chosen Cayuga as a representative event engine, and shown how to express its automata queries and MQO techniques in RUMOR. In this subsection, we compare the performance of our system based on RUMOR with Cayuga. Due to the significant differences in the architecture and implementation platform of both systems, we do not wish to compare their absolute performance. Instead, we follow the experimental approach used in SASE [95], and report on *normalized throughput* obtained as follows: as the query processing load changes from light to heavy in each experiment, we use the throughput for the lightest workload to normalize other measurements (thus they are all under 1). This approach allows us to observe and compare the performance trends of both systems when we vary the values of experimental variables.

Workload 1. In the first query workload, we generate a set of queries of template $\sigma_{\theta_1}(S)$;$_{\theta_2 \wedge \theta_3}$ T, where ; is the Cayuga sequence operator. θ_1 is of form $a[0] = c$, where c is chosen at random between 0 and 999 with the same Zipfian distribution for window lengths. Similarly, θ_3 is of the same form and generated in the same way, but it is



(a) Number of Queries (b) Constant Domain Size (c) Window Length Domain Size (d) Zipf Parameter

Figure 5.8: Event Pattern Query Workload Exercising AN Index and FR Index in Cayuga

evaluated on each T tuple, whereas θ_1 is evaluated on each S tuple. θ_2 is a "duration predicate" in Cayuga terminology – it expresses the window length of this query. Note that this query workload exercises the AN index and FR index in Cayuga, which we described in Section 5.4.3. In particular, the θ_1's of the set of queries we generate can be indexed by an FR index, while the θ_3's can be indexed by an AN index.

We first vary the number of queries. Figure 5.8(a) shows that by expressing AN indexes and FR indexes with m-rules in RUMOR, our system scales very well. Note that even if the predicates θ_1 and θ_3 on each query are quite selective, this is not a trivial query workload – with 100K queries in the system, the input stream of 100K events generates a total of 1.7 million output events.

Next, we vary the constant domain size. Intuitively, the larger the constant domain size, the more selective θ_1 and θ_3 are, and therefore the lower load each query has. The result in Figure 5.8(b) matches our expection.

We then vary the size of window length domain. For a sliding window join query, the larger the window length, the more expensive the query becomes, since it needs to hold more states, and may produce more output tuples. However, the Cayuga sequence operator has the special semantics that when a tuple in the operator state is matched by an incoming tuple from its second input stream, that tuple in the state is deleted. For

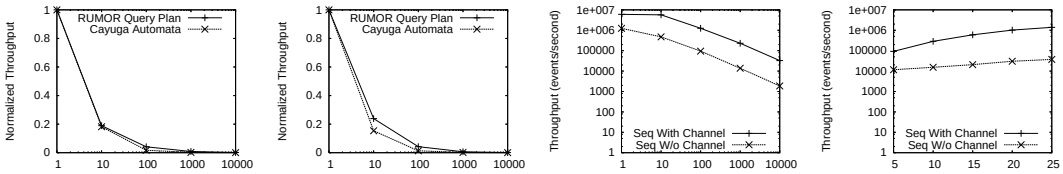
this query workload, most tuples in the operator states are matched by incoming tuples, before they fall out of the query windows and expire. Therefore, as we increase the size of window length domain, creating queries with larger windows, the load of the queries does not significantly increase. This is confirmed by the result shown in Figure 5.8(c).

Finally, we vary the Zipfian parameter value used to generate window lengths and predicate constants, and report the result in Figure 5.8(d). As we increase the Zipfian value, there is increased commonality among the generated queries. When two queries share a common subexpression, both systems will avoid repeated evaluation of the common subexpression. As a result, the throughputs of both systems increase. However, the impact of Zipfian value is not very significant for this query workload (the throughputs of both systems increase by a factor of around 2 when the Zipfian value increases from 1.2 to 2). This is because with the use of an AN index and an FR index, the added value of Common Subexpression Elimination in terms of increasing throughput is little. This is consistent with the conclusion drawn in previous work [29].

Workload 2. We next use a query workload which exercises the AI index technique in Cayuga, an MQO technique which can again be expressed in RUMOR.

The query template we use is $S ;_{\theta_1 \wedge \theta_2} T$, where θ_1 is of form $S.a[0] = T.a[0]$, and θ_2 expresses the window length of this query, as in the previous workload. The AI index in Cayuga indexes for each query the input tuples from S that remain in the state of `;`. This is so that the evaluation of θ_1 on the input T tuples can be sped up.

The result of varying the number of queries is shown in Figure 5.9(a). This query workload is more expensive than the previous one, since each query here does not specify predicates that compare S and T tuples with constants. Intuitively, a query in this workload, which looks for a pair of S, T tuples with the same value on $a[0]$, is a pa-



(a) Number of Sequence Queries (b) Number of μ Queries (c) Number of Queries (d) Channel Capacity

Figure 5.9: Event Pattern Query Workload Exercising AI Index in Cayuga

rameterized version of a query in the previous workload, which looks for a pair of S, T tuples with $a[0]$ values specified by the predicate constants. Hence, when we process each query in this workload, each input S tuple is inserted into the `;` operator state, and each input T tuple probes the operator state. Still, our system is able to maintain high throughput in the presence of 10000 queries.

We also tried a variant of the query template used in this workload: $S\ \mu_{\theta_1 \wedge \theta_2, \theta_3}\ T$. Here μ is the Cayuga iteration operator. θ_3 is the "rebind predicate" which is defined by $T.a[1] > last.a[1]$, where $last.a[1]$ denotes the $a[1]$ value of the last input event that contributes the event pattern being built by this query. Intuitively, each such query looks for an event sequence pattern starting with an S tuple, followed by a sequence of T tuples with increasing $a[1]$ values. The result of varying the number of queries of this template is shown in Figure 5.9(b). The throughput trends of both systems are similar to those in 5.9(a). However, the absolute values are lower, since μ is a more expensive operator to evaluate than `;`.

Similarly to the experiments for the previous workload, we also varied other parameters for this workload, and obtained similar results.

Conclusions of Workload 1 and 2. As we observe in the above results, both systems display similar trends of throughputs when we vary the experimental parameters in the above settings. This is expected since the Cayuga MQO techniques are translated into

RUMOR, and are used by the RUMOR query plans.

Workload 3. The above experiments do not involve channels. Next, we use channels to further share work among queries generated in a variant of Workload 2, with the techniques described in Section 5.4.4. In this workload variant, the query template we use is $S_i \,;_{\theta_1 \wedge \theta_2}\, T$, where the predicate parameter of ; is defined in the same way as above workload. The first input stream S_i's for the generated queries are different streams but *sharable* as defined in Section 5.3.2, and they are encoded with the same channel, denoted as C. The second input stream T is the same for all the queries. By default, there are 10 different S_i streams, referred to as S_1 through S_{10}. For a channel, we define the number of streams encoded by it its *channel capacity*. Thus the default channel capacity of C is 10.

As we have compared the performance of our system with Cayuga in the above workloads, to quantify the benefits of using channels, here we compare the performance of our system with channel against that of our system without using channel. Since both competitors are based on the same software infrastructure, we report absolute throughput in the results below.

We modify the way we generate stream tuples as follows. When we use channels in the query plan, we interleave the generation of tuples from C, encoding S_1 through S_{10}, and the generation of tuples from the channel encoding T alone. Each C tuple belongs to all S_i's. In the case when we do not use channels in the query plan, we use a round robin policy to generate stream tuples: we first generate 10 tuples respectively from S_1 through S_{10}, and then generate a tuple from T. The set of 11 consecutive tuples is referred to as a *round* of tuples. We then repeatedly generate rounds of tuples. To ensure fairness in the comparison, We make the first 10 tuples in every round have the same content. This way, the generated stream used for the query plan with channels,

and the one used for the query plan without channels, have exactly the same content.

Figure 5.9(c) reports the result of varying the number of queries. The throughput of our system using channel is one order of magnitude higher than that of our system not using channel. Note however that this workload generation is optimistic, in that it assumes each C tuple belongs to all S_i's. In a realistic workload, it is usually the case that a channel tuple belongs to a subset of streams the channel encodes, which we measure in Section 5.5.3. Nevertheless, Figure 5.9(c) shows that channel is a very good mechanism for sharing work. Figure 5.9(d) reports the result of varying channel capacity. Clearly, when our system uses channels, the more streams channel C encodes, the higher throughput it achieves. We also peformed experiments on channels with query template $S_i\mu_{\theta_1 \wedge \theta_2, \theta_3}T$, and obtained similar results.

5.5.3 Hybrid Queries with Real Datasets

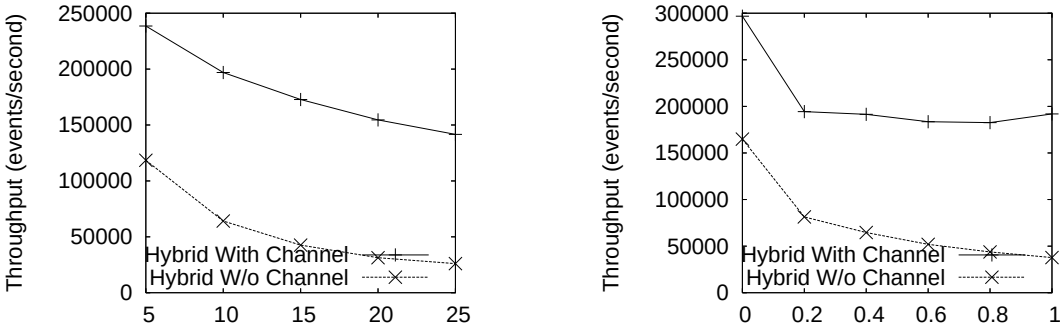
The experimental results on our sythentic benchmark showed that our system is efficient and scalable. We next test our system with hybrid query workloads and two real datasets.

The two performance counter datasets are both collected with the Performance Monitor component of Windows Vista. For the first dataset, called D_1, we chose 104 long running processes on a developer's office machine, and collected the CPU usage of these processes over a 24-hour period of time. For each process, for every second, one stream tuple is recorded for the amount of CPU percentage that process has used in the last second. The second dataset, called D_2, is collected in a similar way, recording the CPU usage of 28 long running processes on a home machine over a day.

For the query workload of this experiment, we choose the set of hybrid query in-

stances of Query 2 with the following modifications to make our query workload more challenging. First, instead of monitoring a particular process with a specified pid in the query, each query monitors *all* processes. Thus, if the input stream contains performance readings for n processes, where $n = 104$ for dataset D_1, one query in our workload corresponds to n instances of Query 2, each monitoring a particular process out of the n processes. Second, for each query, we increase the window length of the sliding window aggregation operator used to smooth the CPU loads of each process from 5 seconds (as in Query 1) to 60 seconds. We still denote the smoothed stream as `SMOOTHED`. Third, we reduce the selectivity of the stopping condition of each query, by setting the stopping condition to $CPU.load > 10$, instead of $CPU.load > 90$ (as in Query 1). As a result, each query in our workload may produce more output tuples than an instance of Query 2. Finally, for the starting condition of each query, we use a parameter $sel \in [0.0, 1.0]$ to control its selectivity. Intuitively, if $sel = 0.0$ for all queries, no stream tuple from `SMOOTHED` will pass the starting conditions, so no event patterns will be produced. With a higher sel value, each `SMOOTHED` tuple may pass a subset of starting conditions of all the queries. We assume these starting conditions are not indexable (since they may have inequality or more complex predicates as in Query 1) , but still use the m-rule s_σ to map all of them to an m-op, denoted as $\sigma_{\{s_1,\cdots,s_n\}}$ as in Figure 5.5(b), which seqentially evaluates them for each input `SMOOTHED` tuple. Without using channels, $\sigma_{\{s_1,\cdots,s_n\}}$ produces one tuple for *each* σ_{s_i} that is satisfied by the current `SMOOTHED` tuple being processed. With channel, however, $\sigma_{\{s_1,\cdots,s_n\}}$ produces one (channel) tuple for *all* σ_{s_i}'s that are satisfied, as in Figure 5.5(c).

We report the throughputs of the RUMOR query plan without using channels (Figure 5.5(b)), and the query plan using channels (Figure 5.5(c)). In Figure 5.10(a), we fix $sel = 0.5$, and vary n, the number of hybrid queries. We are able to achieve very high throughputs in this very challenging hybrid query workload, especially when we use



(a) Number of Hybrid Queries

(b) Selectivity of Starting Conditions

Figure 5.10: Hybrid Query Workload with Real Dataset D_1: Note each query corresponds to 104 instances of Query 2

channels in the query plan. This shows that RUMOR is very effective in sharing work among a set of queries. In Figure 5.10(b), we fix $n = 10$, and vary the selectivity of the starting conditions in the 10 queries. With a higher selectivity, the query plan without using channel experiences a significant degradation in throughput, as more tuples are produced by σ_{s_i}'s, so μ_i's have to do more work. As we expected, after the throughput of the query plan using channels experiences a drop when sel increases from 0.0 to 0.2, that throughput remains stable with larger sel values. This is because for each channel tuple t produced by $\sigma_{\{s_1,\cdots,s_n\}}$, the amount of work for processing it in $\mu_{\{1,\cdots,n\}}$ remains the same, regardless of how many stream tuples t encodes. As such, the more streams that can be encoded by channels in the query plan, the more savings we can obtain compared to the query plan without using channels.

We obtain similar results in processing D_2, and do not report them here.

5.6 Related Work

Stream processing has been well studied as a computational paradigm to continuously process and respond to high-speed data streams [25, 21, 11, 24]. The importance of Multi-Query Optimization, first studied in the context of relational database query processing [76], is recognized in NiagaraCQ [25]. Subsequent work on stream MQO focuses on individual query operator types, which we summarize as follows. We have shown in this chapter how to model all of these MQO techniques as m-rules and m-ops in RUMOR.

The predicate indexing technique is studied in Le Subscribe [33] and CACQ [63]. In [43], Hammad et al. develop techniques to share work among multiple stream join operators which read the same input streams and have the same join predicate but potentially different window specifications. These techniques achieve fairness in scheduling individual queries among them. For multiple aggregate queries with potentially different groupby specifications, Zhang et al. propose to maintain query states in two levels of granularity, such that the aggregation computation performed at the finer-grained level can be shared among queries [99]. For pattern matching queries, state merging and indexing techniques, which we reviewed in Section 5.4.3, are proposed in Cayuga to efficiently evaluate a set of automata queries [29]. Similar indexing techniques, such as Partitioned Active Instance Stack (PAIS), are used in SASE [95].

All of the above MQO techniques attempt to share work among queries reading the *same* input stream(s). Krishnamurthy et al. propose interesting techniques to share work among queries reading *different* streams, where these streams may share tuples of identical content [54, 55]. We generalize their techniques with the concept of channel (Section 5.3), and show how channels can be used to share work among multiple event

pattern queries (Section 5.4.4).

Jiang et al. propose a novel three stage integration model for event and stream processing [51]. Their work takes expressive event processing features other than event pattern queries, such as event consumption modes, into account, which our work has yet to consider. In constrast, our focus is on integrating the MQO techniques for both REs and event engines.

5.7 Remarks

In this chapter, we propose RUMOR, a rule-based MQO framework to express and evaluate query plans that share work among multiple stream queries. RUMOR integrates existing as well as new MQO techniques for both REs and EEs. As a result, we are able to unify REs and EEs, and efficiently process a large number of RE queries, EE queries, and hybrid queries in a single engine.

This work opens up a few avenues for future work. First, it is possible to extend the definition of a channel, allowing it to encode streams of different schemas. Second, as is mentioned in Section 5.3.3, on an input query plan, multiple m-rules can become applicable at the same time. Some conflict resolution strategies will be useful in this case. For example, to reduce or completely eliminate nondeterministic rule applications, rule priorities can be assigned to establish a partial order or a total order in the m-rule set. Furthermore, static analysis techniques can be developed to reason about the confluence of the rule-based query rewrite system. Finally, as in relational query optimization, it is valuable to supplement the rule-based query optimizer with a cost model, such that the optimizer can drive the rule applications based on a cost function, reducing the chance of producing a locally optimal query plan.

CHAPTER 6

XML STREAM JOIN PROCESSING

6.1 Introduction

XML has become the primary standard for data exchange on the Internet and for enterprise applications. The rapid emergence of Web Services in particular has underlined the need to support efficient XML processing in distributed environments. A crucial component of Web Service based architectures are message brokers. They manage large numbers of subscriptions, or queries that express the interest of subscribers — both users and applications. The subscriptions are matched in real-time with *event streams* (or for short, streams) of incoming XML documents, created by publishers like applications behind a Web Service interface, news services, or blog writers. Because of its close relationship to traditional publish/subscribe (pub/sub) systems, we will use the term *XML publish/subscribe system* to refer to this class of message brokers. In the setting of processing XML streams, events and documents are interchangeable terms.

It is crucial for XML pub/sub systems to be both expressive and scalable. Expressiveness refers to the ability of the query language to support a wide variety of queries. The downside of greater expressiveness is that complex queries are more difficult to implement efficiently. For applications like message brokering, an XML pub/sub system has to scale in terms of the number of subscriptions and the stream rate of incoming messages, while providing sufficient functionality to express all relevant subscriptions.

There has been much recent work on XML pub/sub systems that can efficiently process a large number of XML subscriptions over streaming XML documents [7, 26, 31, 37, 41, 69]. These systems support a proper subset of XPath 1.0, typically limited

to forward axes (child and descendant), predicate evaluation and wild card operator $\star$. However, they are unable to express a large class of important queries: queries that correlate *multiple input events* to detect complex patterns in real-time. This class has been recognized as being highly important for event processing [95, 29]. We refer to these queries as *inter-document queries*.

Inter-document queries *join* different XML documents based on values in their nodes, either attributes or text. An inter-document query is capable of joining multiple documents in either the same XML stream, or across multiple streams. For example, for monitoring blogs and news articles, users might be interested in blog postings by the same author or about the same topic that appear within a short time of each other and are above some reputation threshold. Inter-document queries are also building-blocks for more powerful queries like finding all electronics product announcements that "create above-average attention in the blogosphere." In enterprises, related events containing information about the quality of service that customers receive need to be processed to monitor compliance with service level agreements. There has been some emerging work on XQuery stream processing [60, 34]. XQuery can express join queries, but none of the existing systems scales to a large number of concurrently running queries.

Example. We illustrate our approach with a running example. For ease of exposition, we consider processing of a single XML stream `S` that includes book announcements and RSS feed items for blog articles. Our techniques can be easily extended to handle multiple XML streams. Two example documents are shown in Figures 6.1 and 6.2. The superscript of each element node denotes its node id as defined by pre-order traversal of the XML tree. The dashed ovals connected to leaf nodes with dashed lines represent the text values of the leaf nodes in this document.

Table 6.1 shows three example queries. Query Q1 looks for a book announcement

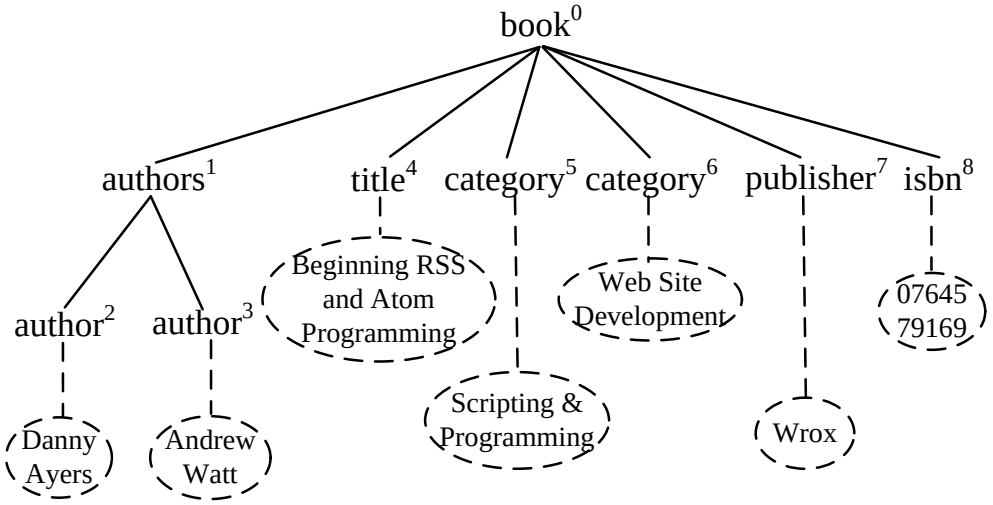


Figure 6.1: A book announcement document $d1$

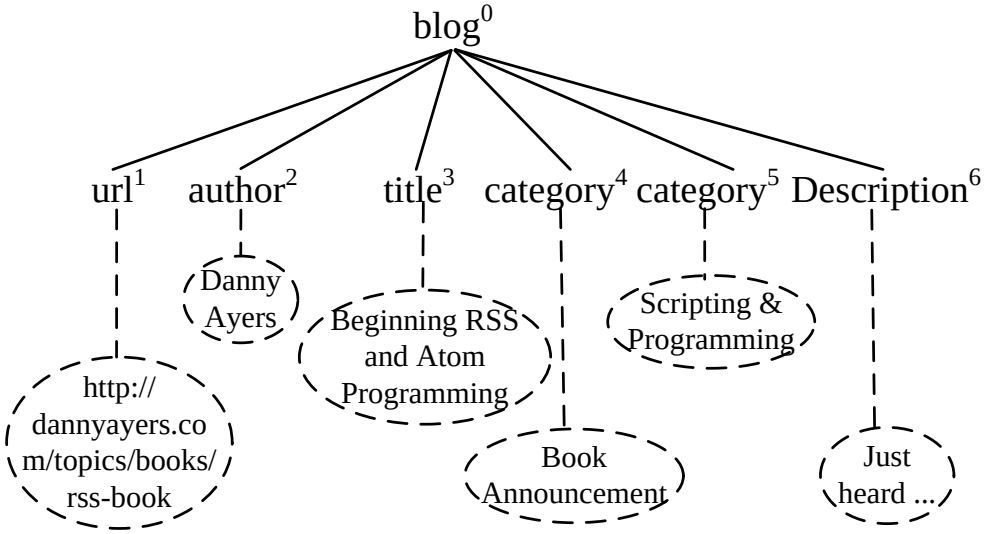


Figure 6.2: A blog article document $d2$

followed by a blog article from one of its authors that promotes this book. Q2 tries to find a book announcement followed by a blog article from one of its authors following up on material in the book. Q3 checks for blog cross-postings.

XML message brokers are used for applications ranging from tens of publishers and subscribers, in small enterprises, to hundreds of thousands of users in Internet scale RSS

Table 6.1: Examples of Inter-Document Queries

Q1	Return a book announcement, followed by a blog article from one of its authors with the same title as the book.
Q2	Return a book announcement, followed by a blog article from one of its authors on the same category as the book.
Q3	Return a pair of blog postings by the same author and with the same title.

Table 6.2: XSCL Formulations of queries in Table 6.1

Q1	S//book->x1[.//author->x2][.//title->x3] FOLLOWED BY{x2=x5 AND x3=x6, T1} S//blog->x4[.//author->x5][.//title->x6]
Q2	S//book->x1[.//author->x2][.//category->x7] FOLLOWED BY{x2=x5 AND x7=x8, T2} S//blog->x4[.//author->x5][.//category->x8]
Q3	S//blog->x4[.//author->x5][.//title->x6] FOLLOWED BY{x5=x5' AND x6=x6', T3} S//blog->x4'[.//author->x5'][.//title->x6']

feed monitoring for blogs and news. Hence an XML pub/sub system has to process anywhere from a few hundred to millions of concurrently active subscriptions for streams that can have high arrival rates. The only way to achieve this kind of scalability is by effective multi-query optimization (MQO).

Unfortunately, MQO for inter-document queries is a very challenging problem. As even the simple queries in Table 6.1 illustrate, the join condition consists both of tree patterns (e.g., to identify the author nodes and title nodes) and node value comparisons

(e.g., equality of author name text for book announcement and blog article). This can create a wide variety of conditions with little apparent commonality. To address this issue, we propose to dissect each query into *tree pattern components* and *value comparison components*. The tree pattern components are expressible in the simpler XPath fragments supported by existing XML pub/sub systems like YFilter [31]. This enables us to leverage existing XML pub/sub technology for efficient discovery of tree pattern components. Unfortunately this does not suffice, because the main performance bottleneck in practice is the evaluation of the value comparison components, as is confirmed by our experimental section.

We show that value comparison components, which have only very limited structure information, almost always can be described by a small number of *query templates*. This is guaranteed for XML documents that have a fairly regular schema, which is common in practice [27], and for documents with a small number of nodes, which is often the case for individual RSS feed items. Even for other XML streams, in practice the number of value comparison components is small, because only a few of the possible comparisons are semantically meaningful. (E.g., it is unlikely that a query would ever compare the author name with the ISBN of a book.) This observation gives us a powerful handle on MQO. Without dissecting join conditions, each different condition would have to be implemented and executed individually, similar to a nested loops join whose outer loop iterates over all queries and whose inner loop evaluates the join predicates. Our dissection approach induces a partitioning of the query set into a small number of equivalence classes, one for each query template. Now we only need a per-template implementation and can take advantage of set-oriented processing of all queries that belong to the same template. By mapping this into a relational join problem, we can take advantage of a wealth of expertise in relational query processing.

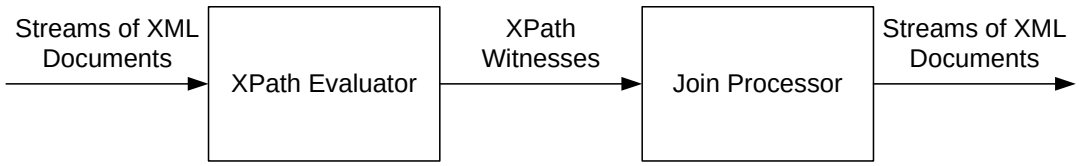


Figure 6.3: Two-Stage Query Processing

The query dissection into tree pattern and value comparison components naturally leads to a *two-stage* approach to query processing. Our system has two major components—the *XPath Evaluator* for processing all tree pattern components and the *Join Processor* for evaluating the value comparison components (see Figure 6.3). For an incoming XML document, first the XPath Evaluator is invoked to evaluate the tree patterns. It produces a set of bindings of variables defined in these patterns. These bindings are referred to as *XPath witnesses*, or *witnesses* for short. Second, the Join Processor uses the witnesses to perform value joins on a per-template basis. In this scheme, the XPath Evaluator can be viewed as an access method or accelerator for efficiently "retrieving" the witnesses for the join processing stage. As mentioned above, we can leverage existing XML pub/sub technology for the XPath Evaluator and hence focus on the Join Processor in this chapter.

Our contributions. The problem we address in this chapter is to efficiently process a large number of continuous inter-document queries against incoming XML streams. Our main contributions are as follows.

- We propose novel *Massively Multi-Query Join Processing* techniques[1] for efficiently evaluating a large number of inter-document queries over streams of XML documents. The key to achieving scalability is to dissect join conditions into tree pattern and value comparison components. This leads to a two-stage processing approach in which both storage and computation can be shared effectively among

[1]This term is grammatically correct since "Massively" refers to "Multi-Query", rather than to "Join."

queries.

- We develop a compact representation for the results of the first processing stage, the tree pattern witnesses produced by the XPath Evaluator, for efficient access during the second processing stage. (Section 6.3)
- We propose a scalable Join Processor for the second stage. The main idea is to map the problem into a relational framework which facilitates sharing of join processing cost across different queries. (Section 6.4)
- We present query optimization techniques for the Join Processor to further improve performance. Here we take advantage of the relational formulation, e.g., for view materialization. (Section 6.5)
- We evaluate the performance of our join processing techniques through an extensive set of experiments in Section 6.6.

We discuss related work in Section 6.7 and conclude in Section 6.8.

6.2 XSCL Query Language

The XPath fragments that form the query language for existing XML pub/sub systems like YFilter are not expressive enough for inter-document queries. It is possible to express these queries in XQuery, but that is a much more general language with many additional features (and complications), which are not relevant for this discussion. Some of the inter-document queries would look unnecessarily complex in XQuery, obscuring the query structure and optimization opportunities.

To be able to express inter-document queries in a natural and compact manner, we define the XML Stream Conjunctive Language, or XSCL for short. XSCL adds join

operators to the XPath operators used by previous XML pub/sub systems. It can be viewed as a fragment of XQuery, i.e., all XSCL queries can be converted into equivalent XQuery expressions. We omit the formal language definition, which is not necessary for grasping the features relevant to this discussion.

Each query in XSCL consists of three clauses: SELECT, FROM and PUBLISH. The SELECT clause specifies how to construct the output XML stream of the query, and is similar to the XQuery RETURN clause. The PUBLISH clause assigns a name to the query's output stream, so that other queries can refer to it as their input. For example, the query "SELECT * FROM blog" outputs every event from input stream blog. This query can be alternatively written as "blog", since in XSCL the SELECT clause can be omitted, defaulting to SELECT *. From a query optimization point of view, the most relevant construct is the FROM clause. It specifies the join condition for the query's input streams, using a variety of operators from two groups—traditional XPath operators and join operators.

XPath operators. Tree patterns in XML documents can be expressed with the same XPath operators that are used by existing XML pub/sub systems. In particular, the following axis operators can be used: / (child), // (descendant), @ (attribute) and [] (predicate). These operators have the usual XPath semantics. We can apply these operators to a particular XML stream `S` by placing the stream name before them. For example, `S`//blog//title outputs the titles of blog articles from stream `S`.

Join operators. In addition to the operators drawn from XPath, XSCL has two join operators, which make it significantly more expressive than the previously used XPath fragments. The join operators are used for inter-document queries. The first, JOIN, is equivalent to the time-based window join operator in the relational data stream processing literature [52]. It has two parameters, *pred* and T, the *join predicate* and *time*

constraint, respectively. The expression A JOIN$\{pred, T\}$ B produces an output event when there is an event produced by expression A and an event produced by expression B occurring within T time units of each other, and they together satisfy predicate *pred*. Subexpressions A and B are composed from XPath operators only. We refer to them as *XPath query blocks*, or *query blocks* for short. We will usually use π to denote a query block. In this chapter we assume *pred* contains only equality predicates. Efficiently processing a large number of inequality predicates is left as future work.

The second join operator, FOLLOWED BY, corresponds to the sequencing operator in event processing systems [22, 29, 95]. It has the same two parameters as JOIN and can be used in the same context. The only difference is that FOLLOWED BY is "forward-looking." Expression A FOLLOWED BY$\{pred, T\}$ B only produces an output result when there is an event produced by expression A followed by (i.e., with timestamp value greater than) an event produced by expression B within T time units, and they together satisfy predicate *pred*.

Notice that the time constraint parameter T requires XML documents to have timestamps. They can be assigned either by the publishers (event sources) or by the XML pub/sub system itself. This choice is application dependent. A detailed discussion on how to manage timestamps is beyond the scope of this work and has been examined in related work [84].

Variable binding construct. In the FROM clause, we can also bind XML element nodes obtained through XPath operators in query blocks to variables through the use of the AS clause. These variables can be referred to in join predicates in the FROM clause, and in the SELECT clause for output construction. (This is similar to SQL's AS clause.)

Examples. Table 6.2 shows the XSCL formulation of the example queries from Table 6.1, using Ti as the window constraint for query Qi. Three points should be noted for the XSCL formulations. First, the semantics of the equality operator in XSCL is defined as equality of the string values of the nodes, where the string value of a node is defined by XPath semantics.

Second, in the FOLLOWED BY predicate $pred$ of an XSCL query, it is possible to apply the standard XPath operators like /, // and [] to variables bound in the query blocks to FOLLOWED BY. However, we can show that any XSCL query can be rewritten into a form where predicates inside the FOLLOWED BY part of the query do not contain any XPath operators and only contain value joins that involve pairs of variables bound in the two input query blocks of FOLLOWED BY. We say that an XSCL query q is in *value-join normal form* if q has this property. In the remainder of this chapter we assume queries are in this normal form. Also, when two variables (in two different queries or in the same query) have exactly the same definition, we assume the two variables are of the same name. Our assumptions are without loss of generality, since these effects can be achieved through rewrite techniques during query insertion. The three queries presented in Table 6.2 fulfill our assumptions.

Third, when the SELECT clause is omitted for a join query, we construct the output XML tree in a default way as follows. We create a new root node and make the root element nodes from the two query blocks its children. For example, for query Q1 each output XML tree has two subtrees under the root, where the first subtree corresponds to the output of XPath expression //book[.//author][.//title] given by the first query block, and the second subtree corresponds to the output of XPath expression //blog[.//author][.//title] given by the second query block.

Expressiveness of XSCL. It is easy to show that XSCL is more expressive than

conjunctive queries [4]. When the join graph of an XSCL query is cyclic, it is therefore NP-hard to find an optimal query evaluation plan (join ordering) in general. Since we would like to process a large number of continuous XSCL queries, this makes our problem even harder. Hence instead of attacking the general conjunctive query processing problem, we propose an efficient solution that is applicable to a very large and practically important subset of the problem instances.

6.3 Stage 1: From XSCL Queries to Value Joins

Recall that the two-stage query processing scheme separates XSCL query processing into XPath tree pattern processing and value join processing. Given a set of input XSCL queries, we take all the (single-document) tree patterns corresponding to query blocks in these queries, and insert them into the XPath Evaluator with the goal of returning *witnesses* that represent single-document variable bindings. For each event e, we first invoke the XPath Evaluator to produce all its witnesses, and then value-join the witnesses from e with witnesses from events earlier in the stream. Due to space constraints, we omit the proof that this two-stage query processing scheme yields correct query results.

In this section we describe the first of the two stages of our multiple XSCL query processing, XPath Processing, and focus on how to efficiently represent the witnesses produced by the XPath Evaluator (Section 6.3.1).

For ease of exposition, we make simplifications to the query structures in the following discussion. First, we consider only XSCL queries with a single FOLLOWED BY operator, where the two corresponding query blocks will match two different XML documents in order to produce a query output. Second, we assume that the predicate of

a FOLLOWED BY operator is a conjunction of simple equality predicates on string values. In the following, each such simple equality predicate is referred to as a *value join predicate* or *value join* for short. We also assume that value joins occur only between leaf nodes of tree patterns. Last, we assume all queries read a single input stream. Our techniques can be extended to handle queries involving multiple FOLLOWED BY or JOIN operators with more complex predicates than conjuncts, and more than one input stream.

6.3.1 XPath Processing and Output Representation

Given an input XML document, the XPath Evaluator can benefit from existing XML pub/sub technology for efficient discovery of tree patterns. How do we represent these witnesses for the second stage value-join processing, while preserving tree structure information in them? One extreme design point for representing XPath witnesses is a relational schema storing each valid combination of all the variable bindings involved in an XPath query block. The other extreme design point would be to completely shred the witnesses into a binary relation of individual bindings of variables, as described below.

For a given XPath query block π, we derive a *variable tree pattern*, which extends the standard notion of an XPath tree pattern [2] by associating each tree node with a variable name. We then create a binary relation for each pair of a parent and a child node in the tree pattern.

This binary relation factors out redundant information. It is analogous to normalization of relational schemas based on functional, multi-value and join dependencies. In addition, the representation for witnesses of one query block will be easy to share among other query blocks that bind to the same XML element nodes. Thus in this chapter we

decided to examine in-depth this way of representing witnesses; a full exploration of this design space is future work.

To reduce the number of relations, instead of using a binary relation for each edge in the variable tree patterns, we use a single relation of four attributes `(var1, var2, node1, node2)` to store the pairs of variable bindings for *all* edges in the variable tree patterns. Each tuple in this relation stores in `node1` and `node2` a binding consisting of a pair of node ids, and this binding corresponds to a pair of variables whose names are stored in `var1` and `var2`. We denote this relation as R_{binW}, which stands for "binary representation of witnesses".

There are other pieces of information that need to be stored for value join processing in the second stage. We encode them in relations as follows. Note that R_{binW} stores bindings of pairs of variables from the currently processed stream document. The id and timestamp of this document are stored in the singleton-relation R_{docTSW} with schema `(docid, timestamp)`. For example, suppose event e_1 in Figure 6.1 has document id $d1$ and timestamp $t1$. When it is the current document being processed, R_{docTSW} contains one tuple `(d1, t1)`. Similarly, binary relation R_{docTS} stores the docid, timestamp pairs of previous documents.

The representation of bindings from previous stream documents is very similar to R_{binW}, and they are all stored in a relation R_{bin}. However, since the bindings could come from different documents, the schema of R_{bin} extends that of R_{binW} with an additional `docid` attribute. Its schema is therefore `(docid, var1, var2, node1, node2)`.

In addition to storing the bindings of variable pairs in the tree pattern, we also need to store the string values of nodes that are bound to variables, so that we can evaluate

the value joins on the string values of these variable bindings in the Join Processor. To store the string values of nodes from the current stream document while avoiding redundancy, we use a binary relation R_{docW} with schema `(node, strVal)` for this purpose. Nodes that are not bound to any variable will not be stored in this relation. Similarly, we store the string values of nodes bound in previous stream documents in a relation R_{doc}. Its schema is `(docid, node, strVal)`, extending that of R_{docW} with a `docid` attribute.

Example continued. Consider again our running example with Queries Q1, Q2, and Q3 shown in Table 6.2. Assume that the document $d1$ shown in Figure 6.1 has been processed. Then Tables 6.4(b) and 6.4(c) show the contents of relations R_{bin} and R_{doc}.

6.4 Stage 2: Processing Value Joins

In this section we propose novel techniques for processing a huge number of value joins. A straightforward way would be to evaluate the FOLLOWED BY operator for each XSCL query separately. This strategy is not scalable for two reasons. First, there is no opportunity for sharing of computation among multiple queries. Second, this one-query-at-a-time processing imposes a specific nested-loop style join strategy, where the "outer loop" iterates over each query, and the "inner loop" completes the join processing for that query. With set-oriented query processing strategies, we can significantly improve performance.

Thus, we would like to group the join processing of multiple queries so that computation can be shared among them, and a more efficient join strategy compared to one-query-at-a-time can be used. However, since join operators in different queries could access different variables and have different join conditions, it seems that set-oriented

processing of multiple queries is extremely hard to achieve.

The key insight here is that with the right query plan, two different queries can still share processing. In this section, we define the notion of query templates, and present the query plans for value-join processing based on query templates. Intuitively, the XSCL queries are partitioned into equivalence classes based on which query templates they belong to. The join processing of all the XSCL queries belonging to the same query template can now be shared. Therefore, instead of performing joins individually for each XSCL query, we now perform a join for each set of XSCL queries belonging to the same query template.[2]

6.4.1 Query Template Based Join Processing

Given an XSCL query Q with two query blocks connected by a FOLLOWED BY operator, such as query Q1 in Table 6.2, we can visualize it as a graph, referred to as a *join graph*, illustrated by Figure 6.4. Each query block is represented by a tree pattern formed by solid, bold edges, referred to as *structural edges*. Each node in the tree pattern is labeled by the name of a variable bound in the corresponding query block in Q. For example, the root node of the left-hand-side tree pattern in Figure 6.4 is labeled by x_1, the name of the variable bound to //book in Q1. There are two types of structural edges, representing child axis and descendant axis. For ease of exposition, we assume only descendant axes are present in the XSCL queries we deal with. For each equality predicate $x = y$ in the FOLLOWED BY predicate of Q, we draw a dashed edge between the two (leaf) nodes corresponding to x and y. We call such an edge a *value join edge*. For example, the value join edge between x_2 and x_5 in Figure 6.4 corresponds to

[2]Mathematically speaking, instead of performing join on the original XSCL query space, we now perform join on the quotient space of the XSCL queries defined by the equivalence relation induced by query templates.

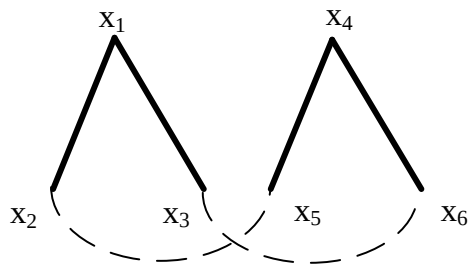


Figure 6.4: Join Graph of Query Q1 in Table 6.2

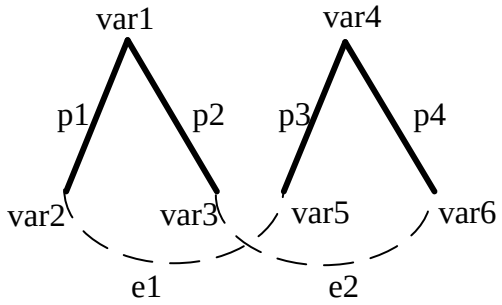


Figure 6.5: Query Template $\mathcal{Q}$ for Q1, Q2 and Q3 in Table 6.2

the join predicate $x_2 = x_5$ in Q1.

A *query template* (or a *template* for short) $\mathcal{Q}$ of Q, is a graph isomorphic to its join graph with different node labels described as follows. Each node u in $\mathcal{Q}$ is labeled by a uniquely named *meta-variable*, whose value is the label of u's corresponding node v in the join graph of Q; i.e., the name of v's corresponding variable in the query Q. Each edge is $\mathcal{Q}$ is also uniquely labeled.

For example, Q1 in Table 6.2 belongs to the query template denoted as $\mathcal{Q}$, which is shown in Figure 6.5. Q2 and Q3 in Table 6.2 also belong to the same query template. The six nodes in this query template are labeled from `var1` to `var6`. The value of the meta-variable `vari` is `xi` for $1 \leq i \leq 6$. The correspondence between nodes and edges in the query template and each query is obvious. For example, edge `p1` connecting `var1` and `var2` in the template corresponds to the structural constraint `x1//x2` in Query Q1.

6.4.2 Sharing Templates With Graph Minor

In Section 6.4.1, we require that the query template $\mathcal{Q}$ of a query Q be isomorphic to its join graph. However, we can show that if we derive a simplified query template $\mathcal{Q}'$ of Q from the graph minor [68] of the join graph of Q through a set of reduction rules below, the join processing result on $\mathcal{Q}'$ will be the same as $\mathcal{Q}$. This enables more queries to share the same query template for join processing.

Given the join graph of Q, we compute its minor via the following reduction rules. First, we recursively remove the leaf nodes that do not participate in any value joins from the join graph. Next, we remove the nodes that are not the descendants of the least common ancestors of the remaining leaf nodes. Finally, we remove all those intermediate nodes that have only one child in the modified join graph. The resulting join graph contains only leaf nodes that participate in value joins, as well as the intermediate nodes that are the least common ancestors of some of the leaf nodes. We derive the query template of Q from the resulting join graph.

The intuition is that since the structural constraints for each individual query block in Q have been evaluated by the XPath Evaluator in Stage 1, the value join processing stage need only check the value constraints, as well as a subset of the structural constraints involving those leaf nodes that satisfy the value constraints.

The number of different query templates depends on the maximum number of value join predicates in the query workload, but not on the number of queries registered with the system, even if these queries have very different tree patterns or seem to equate different nodes. For example, for queries with three value joins in the join predicate of one FOLLOWED BY operator, we show all 16 possible query templates in Figure 6.6. The first 6 templates in the dashed box correspond to the query templates for queries

Table 6.3: Number of Query Templates with respectd to Number of Value Joins

#VJ	#QT(flat schema)	#QT(complex schema)
1	1	1
2	3	3
3	6	16
4	16	$<$230

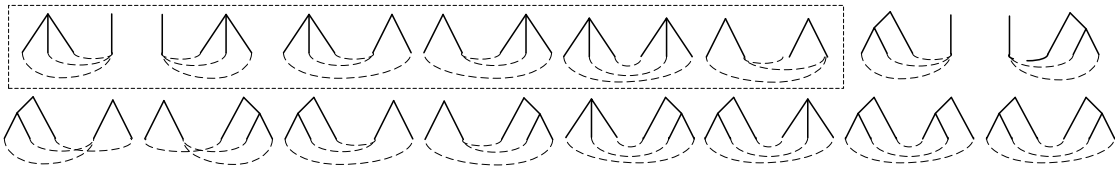


Figure 6.6: 16 Query Templates With 3 Value Joins

defined on a "flat" XML document schema with two tree levels, such as the schema of the blog articles illustrated in Figure 6.2. Table 6.3 shows the relationship between the number of value joins involved in the queries and the number of different query templates for these queries. We leave it as future work to derive a closed-form formula for the exact relationship.

In the remainder of this section, we will explain our join processing techniques based on query templates. Our techniques can be decomposed into two parts. First, we encode all the information needed in join processing as relations, so that we can leverage techniques from relational join processing (Section 6.4.3). Second, for each query template, we create a relational conjunctive query with which we evaluate all XSCL queries belonging to that query template at once (Section 6.4.4). Our query template based join processing algorithm for each document d is given as Algorithm 1.

Table 6.4: Relations involved in Section 6.4.4

(a) R_T for Query Template $\mathcal{Q}$ in Figure 6.5

qid	var1	var2	var3	var4	var5	var6	wl
Q1	x1	x2	x3	x4	x5	x6	T1
Q2	x1	x2	x7	x4	x5	x8	T2
Q3	x4	x5	x6	x4	x5	x6	T3

(b) R_{doc} After Processing $d1$

docid	node	strVal
d1	0	–
d1	2	Danny Ayers
d1	3	Andrew Watt
d1	4	Beginning RSS and Atom Programming
d1	5	Scripting & Programming
d1	6	Web Site Development

(c) R_{bin} After Processing $d1$

docid	var1	var2	node1	node2
d1	x1	x2	0	2
d1	x1	x2	0	3
d1	x1	x3	0	4
d1	x1	x7	0	5
d1	x1	x7	0	6

(d) R_{docW} of $d2$

node	strVal
0	–
2	Danny Ayers
3	Beginning RSS and Atom Programming
4	Book Announcement
5	Scripting & Programming

(e) R_{binW} of $d2$

var1	var2	node1	node2
x4	x5	0	2
x4	x6	0	3
x4	x8	0	4
x4	x8	0	5

(f) Content of R_{outT} After Processing $d2$

qid	docid1	node1	node2	node3	node4	node5	node6	wl
Q1	d1	0	2	4	0	2	3	T1
Q2	d1	0	2	5	0	2	5	T2

Algorithm 1: Join Processing Algorithm

Require: Current stream document d

1: Invoke the XPath Evaluator in d to produce R_{binW}, R_{docW} and R_{docTSW}
2: **for all** query templates $\mathcal{Q}$ in the system **do**
3: Evaluate the corresponding conjunctive query to produce results of XSCL queries belonging to template $\mathcal{Q}$
4: Maintain join state with Algorithm 2

Algorithm 2: Maintain Join State R_{doc}, R_{bin} and R_{docTS}

Require: R_{docW}, R_{binW} and R_{docTSW} produced by the XPath Evaluator when processing the current stream document

1: Set R_{doc} to $R_{doc} \cup (R_{docW} \times \pi_{timestamp}(R_{docTSW}))$
2: Set R_{bin} to $R_{bin} \cup (R_{binW} \times \pi_{timestamp}(R_{docTSW}))$
3: Set R_{docTS} to $R_{docTS} \cup R_{docTSW}$

6.4.3 Representing Join Graphs As Relations

The information needed in join processing includes the join graphs of the XSCL queries, and the XPath witnesses from the current stream document as well as from previous stream documents that participate in the join. We have shown in Section 6.3.1 how to encode XPath witnesses from the current and previous stream documents in relations R_{binW}, R_{docW}, R_{docTSW}, R_{bin}, R_{doc} and R_{docTS}. We now describe how to encode the join graphs of XSCL queries based on query templates as relations.

For each query template $\mathcal{Q}$, we use a relation R_T to encode the join graphs of XSCL queries belonging to this template. The schema contains one attribute `qid` for storing

the query id. Also, it contains one attribute `vari` for each node in the query template labeled by `vari`, the name of a meta-variable. Finally, it contains one attribute `wl` for storing the window length of the join operator. Each query belonging to the template $\mathcal{Q}$ will be encoded as a tuple in relation R_T. For example, the schema and content of relation R_T for the three queries in Table 6.2 belonging to join template $\mathcal{Q}$ in Figure 6.5 is shown in Table 6.4(a).

6.4.4 Conjunctive Query For Each Template

For each XSCL query template $\mathcal{Q}$, we create a relational conjunctive query, denoted as CQ_T, so that the XSCL queries belonging to $\mathcal{Q}$ can be evaluated all at once in CQ_T.

We present the conjunctive queries in Datalog. For a given query template $\mathcal{Q}$, here is how we create CQ_T. For each value join edge in template $\mathcal{Q}$, there is a copy of R_{doc} and R_{docW} joined on their string value attributes in the body of CQ_T. For example, for edge e_1 of query template $\mathcal{Q}$ in Figure 6.5, we put a copy of R_{doc}`(docid, node2, strVal)` and R_{docW}`(node5, strVal)` joined on string value `strVal` in the body of CQ_T. For each structural edge in the query template, we put a copy of R_{bin} or R_{binW} in CQ_T body, depending on whether this edge appears on the LHS or RHS tree pattern in the query template. We do not need to evaluate in the body of CQ_T the tree pattern parts of the XSCL queries, since these structural constraints have been evaluated in the XPath Evaluator, and their results have been stored in R_{binW} and R_{bin}. For example, for edge $p1$ of query template $\mathcal{Q}$ in Figure 6.5, we put a copy of R_{binW}`(docid, var1, var2, node1, node2)` in the body of CQ_T. This completes the construction of CQ_T body.

The head of the conjunctive query CQ_T is a relation denoted as R_{outT}, whose

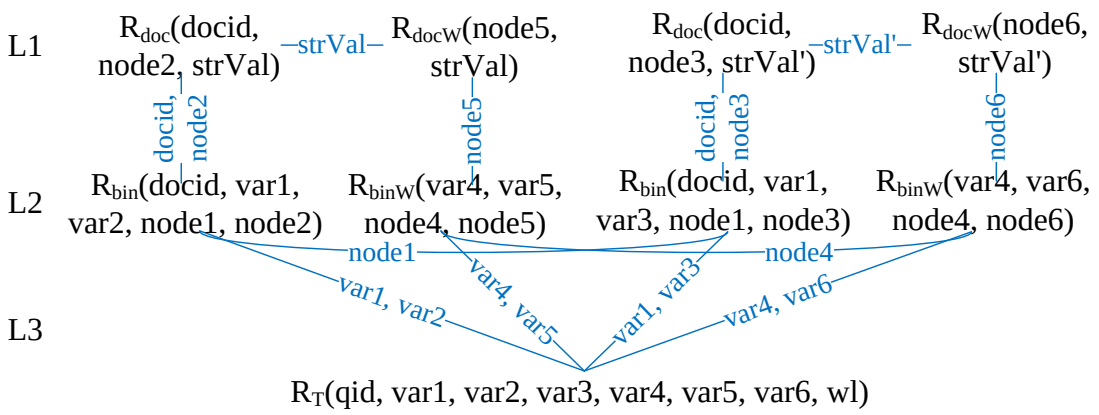


Figure 6.7: Relational Conjunctive Query CQ_T For XSCL Query Template $\mathcal{Q}$ in Figure 6.5

schema contains `qid`, `docid1`, `wl`, as well as one attribute for each node involved in the conjunctive query. For example, the schema of R_{outT} for query template $\mathcal{Q}$ in Figure 6.5 is `(qid, docid1, node1, node2, node3, node4, node5, node6, wl)`, where `nodei` stores the binding node id of an XSCL query variable whose named is stored as value in `vari` in the template. For each tuple in this relation, `node1` through `node3` values come from document `docid1`. `node4` through `node6` values come from the current document.

Below we give the Datalog representation of the conjunctive query for query template $\mathcal{Q}$ in Figure 6.5.

$$
\begin{aligned}
&R_{outT}(qid, docid, node1, node2, node3, node4, node5, node6, wl) :- \\
&R_{doc}(docid, node2, strVal), R_{bin}(docid, var1, var2, node1, node2), \\
&R_{docW}(node5, strVal), R_{binW}(var4, var5, node4, node5), \\
&R_{doc}(docid, node3, strVal'), R_{bin}(docid, var1, var3, node1, node3), \\
&R_{docW}(node6, strVal'), R_{binW}(var4, var6, node4, node6), \\
&R_T(qid, var1, var2, var3, var4, var5, var6, wl)
\end{aligned}
$$

Algorithm 3: Producing Query Results From R_{outT}

Require: Input relations R_{outT}, R_{docTSW} and R_{docTS}

1: Let the single tuple in R_{docTSW} be $d2$
2: **for all** tuples a in R_{outT} **do**
3: Find a tuple $d1$ in R_{docTS} with $d1.docid = a.docid1$
4: **if** $0 < d2.timestamp - d1.timestamp \leq a.wl$ **then**
5: Construct an output XML document for the query with id $a.qid$ based on the specification of its SELECT clause

This conjunctive query CQ_T is visualized in Figure 6.7. In this figure, each node is a relation in the body of CQ_T. There is an edge between two relations, if there is a join between them. The edge is labeled by the set of attributes on which the two relations are joined. In the visualization of the conjunctive query, we place the relations in three levels, denoted as L1, L2 and L3. The relations in level L1 are copies of R_{doc} and R_{docW}. The relations in L2 are copies of R_{bin} and R_{binW}. In level L3, there is always only one relation R_T for the query template $\mathcal{Q}$. The relations in level L1, L2 and L3 are joined together to produce R_{outT}.

To produce final query outputs from R_{outT}, we invoke Algorithm 3, which iterates over tuples in R_{outT}. For each tuple, we first make sure that the temporal constraint of its corresponding query is satisfied (Line 4). Note that the temporal constraint we check in Algorithm 3 corresponds to that for FOLLOWED BY operator. If the temporal constraint is satisfied, we then produce an output XML document according to the specification of the SELECT clause in that query. This process of producing query results from R_{outT} is straightforward. We therefore do not discuss it further and focus only on the conjunctive query CQ_T that produces relation R_{outT} for each query template $\mathcal{Q}$.

After query results have been generated for the current document, in Line 4 of Algorithm 1, we maintain the join state consisting of relations R_{doc}, R_{bin} and R_{docTS} with Algorithm 2. Afterwards, we can discard the relations R_{docW}, R_{binW} and R_{docTSW}, and start processing the next stream document.

Query Processing Example

Let us now walk through the query processing steps for queries Q1, Q2, Q3 in Table 6.2 against the sequence of two documents $d1$ and $d2$ shown in Figure 6.1 and 6.2, which have timestamps $t1$ and $t2$ ($t1 < t2$) respectively.

When document $d1$ comes into the system, since R_{doc} and R_{bin} are initially empty, $d1$ does not produce any query result. R_{docW}, R_{binW} and R_{docTSW} are then merged into R_{doc}, R_{bin} and R_{docTS} respectively, with the `docid` value of each new tuple in R_{doc} and R_{bin} set to $d1$. The content of R_{doc} and R_{bin} at the end of processing this document is shown respectively in Table 6.4(b) and 6.4(c). R_{docTS} contains only one tuple, $\{(d1, t1)\}$.

When document $d2$ arrives, we show the content of R_{docW} and R_{binW} produced by the XPath Evaluator in Table 6.4(d) and 6.4(e). R_{docTSW} contains one tuple $\{(d2, t2)\}$.

Now we want to join R_{doc}, $R_{bin}, R_{docW}, R_{binW}$, and R_T to produce R_{outT}. The content of R_{outT} is shown in Table 6.4(f).

Finally, we invoke Algorithm 3 to produce one output XML document each for query Q1 and Q2. According to XSCL semantics, the two output XML documents produced by Q1 and Q2 have exactly the same content. The root of the output document has two subtrees, where the first subtree corresponds to the subtree rooted at the `book` element

in $d1$, and the second subtree corresponds to the subtree rooted at the `blog` element in $d2$.

6.5 Query Optimization

We have presented the basic ideas of query template based join processing in Section 6.4. The result of these techniques, Algorithm 1, evaluates the conjunctive queries for different templates independently as is shown in Line 2. It therefore leaves much room for sharing computation among these query templates. Also, join processing for the current XML event on the stream might benefit from remembering the results of processing previous XML events. In this section, we propose view materialization as the solution to both these issues.

So far we have assumed that we keep as join state only R_{doc}, R_{bin} and R_{docTS}. We have not considered materializing any intermediate join results for the conjunctive query CQ_T of a query template $\mathcal{Q}$. We now would like to explore the view materialization spectrum with respect to join processing cost.

Let $R_{\hat{L}}$ denote the result of joining R_{doc} and R_{bin}. In one extreme of the spectrum, adopted by the Algorithm 1, we do not materialize $R_{\hat{L}}$, and instead compute it from R_{doc} and R_{bin} for each incoming document. This is likely to result in redundant computation in the join processing. In the other extreme of the spectrum, we can try to materialize the entire $R_{\hat{L}}$, and keep it up to date after processing each incoming document. The materialization of $R_{\hat{L}}$ makes the join processing for each input document less expensive. However, the view maintenance cost of $R_{\hat{L}}$ is likely to be high, since in order to maintain $R_{\hat{L}}$ for each incoming document, we need to first join R_{binW} and R_{docW} together, whose result is denoted as $R_{\hat{R}}$, and then merge $R_{\hat{R}}$ into the existing $R_{\hat{L}}$. Although R_{docW} will

be small for each incoming document, the size of R_{binW} could be proportional to the number of XSCL queries in the system, and therefore the join result could be very large. Also, it may not be worth maintaining the entire $R_{\hat{L}}$, if we do not use such a materialized result in its entirety in processing future documents. We would therefore like to find a sweet spot in the materialization spectrum to minimize the sum of join processing and view maintenance costs.

Determining how much of $R_{\hat{L}}$ to materialize requires a careful study of how $R_{\hat{L}}$ is used in query processing. The schema of $R_{\hat{L}}$ is `(docid1, var1, var2, node1, node2, strVal)`, where variables `var1` and `var2` bind respectively to nodes `node1` and `node2` in document `docid1`, and `node1` is an ancestor of `node2`. Also, `strVal` is a string value corresponding to node `node2`. Recall this is because we assumed that value joins only happen at tree pattern leaf nodes; that is, R_{doc} and R_{bin} are joined on R_{doc}.`node` = R_{bin}.`node2`, and therefore `strVal` in the result corresponds to the string value of node `node2`.

Note that for each incoming document, we usually do not have to access *all* the tuples in $R_{\hat{L}}$. Instead, we only need to access those tuples whose string values appear in the nodes from the current stream document that are bound to variables. In other words, we will only access those tuples in $R_{\hat{L}}$ whose string values are in the result of $R_{docW} :_{\texttt{strVal}} R_{doc}$. Formally, we denote this subset of $R_{\hat{L}}$ as R_L, defined by $R_{docW} :_{\texttt{strVal}} (R_{doc} :_{\texttt{node=node2}} R_{bin})$. If we could materialize this part of $R_{\hat{L}}$, then we could save the costs of the joins that produce them in the join processing for conjunctive query CQ_T's. Also, this observation is symmetric between $R_{\hat{L}}$ and $R_{\hat{R}}$. That is, those tuples in $R_{\hat{R}}$ whose string values correspond to some nodes in $R_{\hat{L}}$ will be accessed and participate in other joins. This means we will have to compute those parts of $R_{\hat{R}}$. Formally, the subset of $R_{\hat{R}}$ that needs to be computed is

$R_R \equiv R_{doc} :_{\texttt{strVal}} (R_{docW} :_{\texttt{node2=node}} R_{binW})$. In sum, only the tuples in R_L and R_R will participate in conjunctive query processing.

For each incoming XML event, we cannot avoid the cost of computing R_R. However, it is possible to reduce the cost of computing R_L through materialization of join results for previous events. To do so, we break up $R_{\hat{L}}$ into slices, where each slice is a set of tuples produced by the join of tuples in R_{doc} with a certain string value and R_{bin}. Specifically, we keep a "view cache" of slices in $R_{\hat{L}}$, denoted as VC, where each cache entry is keyed on a string value s, and stores in the value component a relation $R_{L,s}$, computed by $\mathcal{E}_{L,s} \equiv \sigma_{\texttt{strVal}=s}(R_{doc}) :_{\texttt{node=node2}} R_{bin}$. Similarly, we define $\mathcal{E}_{R,s}$ to be $\sigma_{\texttt{strVal}=s}(R_{docW}) :_{\texttt{node=node2}} R_{binW}$.

Whenever we perform a join between the set of tuples in R_{doc} with a certain string value s and R_{bin}, we first look up the view cache with search key s, to see whether it has been materialized. The size of the view cache can be set according to the memory constraint of the system. Cached entries can be replaced by a cache replacement policy appropriate for the workload, such as LRU.

We incorporate the materialization based optimization above into Algorithm 1 to produce an improved algorithm, Algorithm 4. Essentially, Line 2 through Line 8 are newly added to compute the slices of R_L and R_R, in order to reduce the query processing cost of Line 10. The computation of slices of R_L benefits from remembering the partial result of processing previous XML events, in particular, slices of $R_{\hat{L}}$. The union of these computed slices of R_L (resp. R_R) gives the result of the entire R_L (resp. R_R).

We then evaluate the conjunctive query for each query template in Line 9 – 10. Note that we no longer need to access $R_{bin}, R_{binW}, R_{doc}$ and R_{docW}. Instead, we access only R_L and R_R computed above. This enables sharing of join processing among different

query templates. For example, to process query template $\mathcal{Q}$ in Figure 6.5, we modify the conjunctive query CQ_T presented in Section 6.4.4 into the following query which accesses only R_L, R_R and R_T.

$$
\begin{aligned}
&R_{outT}(qid, docid, node1, node2, node3, node4, node5, node6, wl) :- \\
&R_L(docid, var1, var2, node1, node2, s), \\
&R_R(var4, var5, node4, node5, s), \\
&R_L(docid, var1, var3, node1, node3, s'), \\
&R_R(var4, var6, node4, node6, s'), \\
&R_T(qid, var1, var2, var3, var4, var5, var6, wl)
\end{aligned}
$$

Finally, we maintain the join state and view cache in Line 11 – 12 of Algorithm 4.

6.6 Performance Evaluation

We measure the performance of join processing and our optimization techniques at two levels. We generate a technical benchmark through synthetically generated data of different document schema complexity, and we also measure the performance of our techniques on real RSS data. We have written an XSCL translator, which translates XSCL queries into SQL queries that correspond to the relational conjunctive queries described in Section 6.4. These SQL queries are then evaluated on an SQL engine. We choose Microsoft SQL Server 2005 Standard Edition in the experiments as the Join Processor. All experiments were run on a Dual Core 3.6 GHz Pentium D PC with 3.5 GB of RAM. The operating system is Windows XP Professional. We repeat each experiment 10 times. The standard deviation in all runs was well below 1%; we therefore report only averages, omitting error bars from the graphs.

Algorithm 4: Improved Join Processing Algorithm With View Cache

Require: Current stream document d

1: Invoke the XPath Evaluator in d to produce R_{binW}, R_{docW} and R_{docTSW}
2: Semi-join R_{docW} with R_{doc} on `strVal` to obtain a set STR of common string values
3: **for all** distinct string values s in STR **do**
4: **if** there is an entry with key s in view cache VC **then**
5: Set relation $R_{L,s}$ to the value component of the entry
6: **else**
7: Compute relation $R_{L,s}$ by $\mathcal{E}_{L,s}$, and insert an entry into VC with key s, and value $R_{L,s}$
8: Compute relation $R_{R,s}$ by $\mathcal{E}_{R,s}$
9: **for all** query templates $\mathcal{Q}$ in the system **do**
10: Evaluate the corresponding conjunctive query CQ_T, with $R_{L,s}$'s and $R_{R,s}$'s computed above
11: Maintain join state with Algorithm 2
12: Maintain VC with Algorithm 5

6.6.1 Technical Benchmark

In this first set of experiments, we evaluate a set of XSCL queries that join two fixed input documents. We compare the performance of our join processing algorithm from Section 6.4, which we denote as MMQJP in the figures, with a naive approach which sequentially evaluates the FOLLOWED BY operator in each XSCL query, denoted as Sequential. We run this experiment on XML documents with different complexity in their schema.

Algorithm 5: Maintain View Cache VC

Require: Set STR of common string values in R_{doc} and R_{docW}

Require: The $R_{L,s}$'s and $R_{R,s}$'s computed when processing the current document

1: **for all** string values s in STR **do**

2: Set $R_{L,s}$ to $R_{L,s} \cup R_{R,s}$

3: Insert/Update the cache entry keyed on s with value $R_{L,s}$

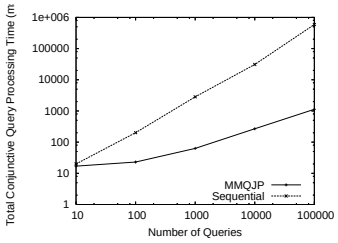


Figure 6.8: Performance on Simple Document Schema

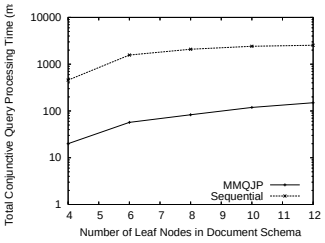


Figure 6.9: Performance on Simple Document Schema

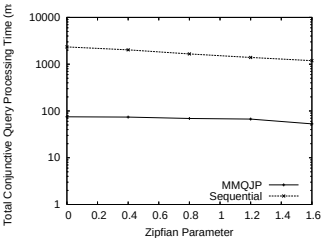


Figure 6.10: Performance on Simple Document Schema

Two-Level Document Schema. We first choose a document schema that models the schema of an RSS feed item, shown by the example in Figure 6.2. The schema has only two levels, where all leaves are children of the root. Let N be the number of leaves in the schema. parameters in this experiment and their default values are shown in Table 6.6.1.

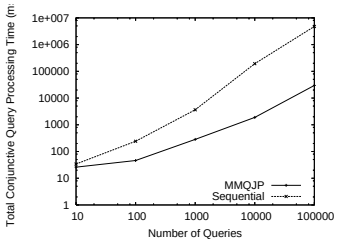


Figure 6.11: Performance on Complex Document Schema

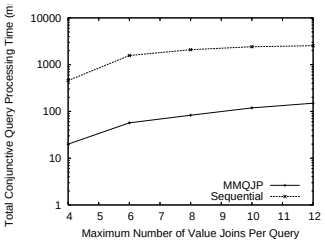


Figure 6.12: Performance on Complex Document Schema

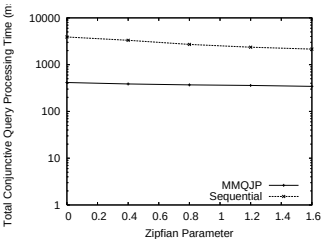


Figure 6.13: Performance on Complex Document Schema

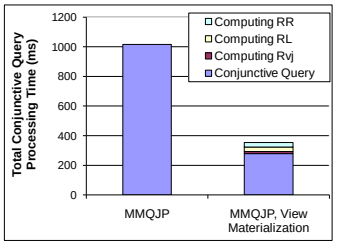


Figure 6.14: View Materialization on Simple Document Schema

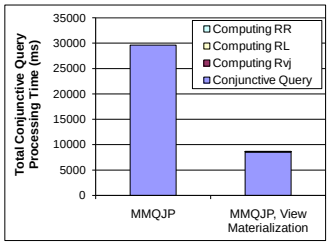


Figure 6.15: View Materialization on Complex Document Schema

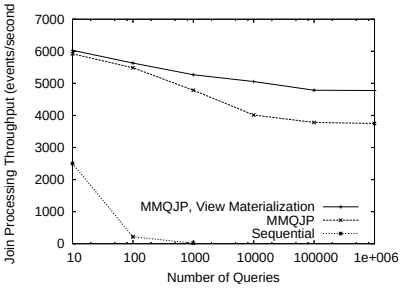


Figure 6.16: Performance on RSS Stream Processing

We then manually compose two documents conforming to this schema, referred to as d_1 and d_2. The root node in d_1 is denoted as n_0, and the N leaf nodes in d_1 are denoted as n_1 through n_N. Similarly, the root node in d_2 is denoted as n'_0, and the N leaf nodes in d_2 are denoted as n'_1 through n'_N. These two documents have the property that all leaf nodes in each document have different string values, but each leaf node n_i in $d1$ has the same string value as the leaf node n'_i in the corresponding position in $d2$, for $1 \leq i \leq N$.

Since our focus is measuring the performance of the join processor, we need to compute $R_{doc}, R_{docW}, R_{bin}$ and R_{binW} as the inputs to join processing. Given the properties of the two documents, we compute these tables as follows. We insert N tuples into R_{doc} corresponding to the N leaves in d_1, where each tuple stores the information of node ID and the string value of a particular leaf in d_1. R_{bin} also contains N tuples, where each tuple corresponds to a particular parent, child pair in d_1. Similarly, we load information of d_2 into R_{docW} and R_{binW}. Note that the tables generated above are guaranteed to be supersets of the results returned by the XPath Evaluator on any number of XPath query blocks. We therefore do not need to invoke the XPath Evaluator in this experiment.

We generate each XSCL query by first selecting a set of variables bound in the LHS and RHS tree patterns of that query in the following way. We randomly pick an integer value k from 1 to N with a Zipfian distribution. For the LHS tree pattern, there are k

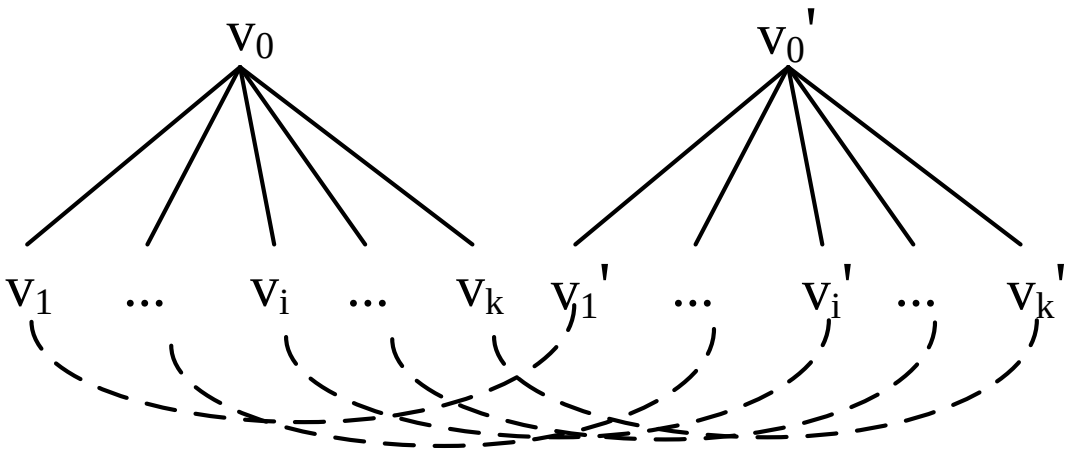


Figure 6.17: Random Generation of XSCL Queries

variables bound to the leaf nodes in the document schema, denoted as v_1 through v_k, as well as a variable v_0 bound to the root node. v_0 is bound only to root node n_0 in document d_1. The k variables $v_i(1 \leq i \leq k)$, are mapped to k different leaf nodes $n_j(1 \leq j \leq N)$ in document d_1 chosen uniformly at random. Similarly, there are k variables v'_1 through v'_k bound to leaf nodes for RHS tree pattern, as well as a variable v'_0 bound to the root. v'_0 is bound only to n'_0 in document d_2. The k variables $v'_i(1 \leq i \leq k)$ are randomly bound to k different leaf nodes $n'_j(1 \leq j \leq N)$ in d_2. We now generate k value joins for this query, where the i^{th} join has a string value equality predicate $v_i = v'_i$. This finishes the construction of query Q. The query construction is shown in Figure 6.17. Observe that based on this query generation approach, the maximum number of query templates in our join techniques is exactly N, regardless of the actual number of XSCL queries generated.

First, we vary the number of XSCL queries, and show the result in Figure 6.8. When the number of queries is small, the performance of MMQJP and sequential evaluation does not differ much. However, MMQJP gains more than two orders of magnitude improvement when the number of queries is large.

Table 6.5: Parameters (default values)

Variable	**Default Value**
Number of XSCL queries	1000
Number of leaves in document schema	6
Zipfian parameter	0.8

We then vary N, the number of leaf nodes in the schema. The result is shown in Figure 6.9. In the way we generate XSCL queries, increasing N will result in more query templates in MMQJP. The time cost of both approaches is about 6 times larger at $N = 12$ compared to $N = 4$; recall from Section 6.4 that the complexity of the query template does not increase linearly with N.

We also vary the Zipfian parameter for generating k for each query (queries with smaller k values are more likely to be generated), and show the results in Figure 6.10. Parameter k has little impact on the performance of MMQJP, since the number of query templates remain the same under these Zipfian values[3]. On the other hand, the performance of sequential evaluation improves by a factor of 2 when the Zipfian value increases from 0.0 to 1.6, because the queries are much simpler at a higher value of the parameter of the Zipf distribution.

Three-Level Document Schema. We repeat the same set of experiments on a more complex document schema. This schema has three levels of tree nodes, where the root and the intermediate nodes all have a branching factor of 4, resulting in 16 leaf nodes in this schema. As in the previous setting, we manually compose two documents d_1 and d_2 conforming to this schema, with the property that the string values of the leaf nodes in the corresponding positions of the two documents are identical.

[3]Only when the Zipfian distribution is extremely skewed, some query templates involving many value joins will not occur.

In this setting, we have a new parameter K, denoting the maximum number of value joins per query. Its default value is 4. To generate each query, we first randomly pick a value k from 1 to K with Zipfian distribution. As in the previous setting, for the LHS tree pattern, there are k variables v_1 through v_k bound to leaf nodes in the document schema. We pick uniformly at random k different leaf nodes from d_1 to be bound to these k variables. variable v_0 in LHS pattern is bound to the root node of document d_1. Now, to form a more complex tree pattern compared to the previous setting, the nodes in the intermediate level of the document schema that are along the paths between the root node and the leaf nodes bounded by v_1 through v_k will be bounded by additional variables in the LHS tree pattern. This adds additional structural joins in the conjunctive query for each query template. The construction for RHS tree pattern is similar. Finally, we generate k value join predicates for the XSCL query, where the i^{th} predicate is $v_i = v_i'$.

In this setup, we vary the number of queries, the maximum number of value joins per query, and the Zipfian parameter for generating k. The results are shown Figures 6.11, 6.12 and 6.13, respectively.

When we vary the number of queries, the time cost of both approaches grows more than linearly. This is because as the number of queries grows, more query templates are involved. In MMQJP, the number of query templates increases from 6 to 22 when the number of queries grow from 10 to 100000. Still, MMQJP outperforms sequential evaluation by two orders of magnitude when there are 100000 queries.

When we vary K, the maximum number of value joins per XSCL query, we see that the time cost of MMQJP grows faster than sequential evaluation. This is because MMQJP is affected more significantly by the increasing number of query templates. The numbers of query templates are 2, 6, 20 and 39 for $K = 2, 3, 4$ and 5, respectively.

Varying the Zipfian parameter in this setting has a larger impact on the performance of sequential evaluation compared to MMQJP, because similar as in the previous scenario the numbers of query templates stay constant (around 20), whereas many actual queries have a simpler structure.

6.6.2 Query Optimization

We presented query optimization techniques based on view materialization in Section 6.5. We now evaluate its effectiveness based on the synthetic workload described in the previous section. Since we are interacting with the database engine on the level of SQL, it is difficult to cache slices of $R_{\hat{L}}$ as was described in Section 6.5. Therefore, given the input $R_{bin}, R_{binW}, R_{doc}$ and R_{docW} to the Join Processor, we materialize the following relations:

$$R_{vj}(n1, n1', s) :- R_{doc}(d1, n1, s), R_{docW}(n1', s)$$
$$R_L(d1, v1, v2, n1, n2, s) :- R_{vj}(n1, n1', s), R_{bin}(d1, v1, v2, n1, n2)$$
$$R_R(v1, v2, n1', n2', s) :- R_{vj}(n1, n1', s), R_{binW}(v1, v2, n1', n2')$$

We then evaluate the conjunctive query CQ_T for each query template $\mathcal{Q}$ based only on R_L and R_R, and we compare the join processing cost of MMQJP without view materialization and the cost of MMQJP with view materialization. For the latter, we also measure the time cost of computing R_{vj}, R_L and R_R, respectively.

The experiments are performed on both the two-level and the three-level document schema. We use the default values for all parameters above, except that we set the number of queries to 100000. The results on the two-level and the three-level document schema and shown respectively in Figure 6.14 and 6.15.

Since according to the experiment setup, R_{bin} and R_{doc} only contain information for a single document, d_1, the materialization costs of R_{vj}, R_L and R_R are small com-

pared to the join processing cost. However, we expect that the materialization cost of R_L could potentially be large in real stream settings, since R_{bin} might contain many tuples produced by the XPath Evaluator from previous events. Therefore the benefit of materializing slices of $R_{\hat{L}}$ for computing R_L, instead of recomputing R_L from scratch when processing each event should be significant. Also, in this experiment, we assume we can afford the space to materialize the entire R_L. In practice we may only be able to materialize some slices of R_L, in which case view cache replacement policies may be involved, as was mentioned in Section 6.5.

The results show great benefits from evaluating conjunctive queries by first materializing these relations. This is especially true for the case of the three-level document schema, where we have significantly more query templates compared to the two-level schema (22 templates for complex schema versus 6 for the simple schema). Materializing these relations enables sharing of computation among the conjunctive queries for different query templates; therefore, the more query templates we have, the more benefits we receive from view materialization.

6.6.3 XSCL Queries over RSS Streams

We evaluate the performance of MMQJP and sequential evaluation of XSCL queries over (RSS and Atom) feed streams. The feeds we use in this experiment are collected from 418 channels over a period of time from June to October in 2006. There are a total of 225K items in the feed. Each feed item has a simple document schema similar to the schema in Figure 6.2. Specifically, it has five leaf nodes tagged `item_url`, `channel_url`, `title`, `timestamp` and `description`.

We randomly generate queries in the same way as in Section 6.6.1. We assign a

time window of ∞ to all the generated queries. This means in processing the 225K feed items, no feed item will be discarded from the join state.

Processing XSCL queries over streams involves both the XPath Evaluator and the Join Processor. We evaluated the XPath expressions corresponding to the XPath query blocks we generated on YFilter, an instance of the XPath Evaluator, and we found the time cost of XPath processing over the entire stream in YFilter is about 15 seconds, which is significantly less than the time cost in join processing (using either MMQJP or sequential evaluation).[4] Therefore, the join processing is the bottleneck of the overall XSCL query processing, and in the following text we focus on measuring the cost of join processing.

To run stream processing experiments on a relational database, we perform the following operations for each feed item. First, we issue bulk load statements to load the data of the current feed item into R_{binW} and R_{docW}. The way we generate R_{binW} and R_{docW} is similar to the way we described in Section 6.6.1. We do not include the loading cost in our numbers, since that cost will be negligible in a real main memory based implementation. Next, we evaluate the conjunctive queries, and measure their costs. We then move data from R_{binW} to R_{bin}, and R_{docW} to R_{doc} with SQL statements, however for the same reason as before we also do not include this cost in our overall numbers. We run MMQJP with and without view materialization and also compare to Sequential. We report the total time cost of evaluating conjunctive queries over all the items in the web feed stream.

According to the this setup, there are five different query templates in MMQJP. For each feed item, SQL Server needs to evaluate the SQL queries corresponding to CQ_T for each $\mathcal{Q}$ defined in Section 6.4.4. This means over a stream of S events, the number

[4]The YFilter implementation we use is based on Java; still its XPath evaluation cost is much smaller compared the join processing cost measured in SQL Server.

of queries to evaluate for MMQJP will be $5S$. However, since there is a fixed overhead in the order of tens of milliseconds in submitting an SQL query to a secondary-storage based relational database engine, a measurement of the total cost of evaluating these $5S$ SQL queries will not reflect the real throughput of a publish/subscribe system. Therefore, instead of evaluating the conjunctive queries for query templates once for each feed item, we batch the join processing by loading a set of feed items into R_{binW} and R_{docW} at one time and perform the joins. This significantly reduces the total number of SQL queries to evaluate.

The throughput of MMQJP compared to sequential evaluation while varying the number of queries is reported in Figure 6.16. MMQJP demonstrates impressive throughput with a large number of queries. View materialization helps further by enabling sharing of computation among different query templates. The throughput of MMQJP with or without view materialization stays flat after the number of queries grow beyond 10000, since there are only thousands of distinct queries according to our query generation scheme — after generating 10000 queries, almost all queries generated later on are duplicates. This is consistent with our assumption about the workload. Note that we recompute R_L from scratch for every batch in this experiment, since we did not materialize slices of $R_{\hat{L}}$. Therefore, we expect the throughput of MMQJP with view materialization to be even higher if that is done. The experimental results where we vary the parameter of the Zipf distribution are similar, and we thus omit them from this chapter.

6.7 Related Work

XML Stream Processing. Our work is the first to address both expressiveness in

query language and scalability in system throughput for XML publish/subscribe systems. There has been a large body of work on XML query processing, each addressing parts of these challenges [26, 37, 31, 61]. YFilter [31], XPush [41] and XSQ [69] are based on variants of finite-state automata, and support a significant portion of XPath 1.0 for stream processing. They however do not support queries joining multiple documents or streams. Other XML pub/sub work on more expressive XML query languages has focused on specific optimizations for a small number of queries [60, 53, 19]. Our MMQJP techniques can potentially be combined with these optimization techniques in an XML publish/subscribe system. Examining this is part of our future work.

Other Related Work. Traditional pub/sub systems [6, 98, 33] sacrifice expressiveness to achieve high performance. For example, Le Subscribe [33] is a highly scalable pub/sub system. More recently, Cayuga [29] and SASE [95] propose stateful publish/subscribe systems for complex relational event processing. Data streams have attracted considerable attention in the database community in recent years. Existing DSMSes concentrate on processing of complex relational queries and do not explore multi-query optimization in depth [21, 67, 24, 3].

6.8 Remarks

We have presented Massively Multi-Query Join Processing (MMQJP) techniques, which efficiently process large numbers of continuous inter-document queries over XML streams. Though not elaborated in this chapter, it is easy to see that our approach is also applicable to continuous query processing over relational streams.

CHAPTER 7

CONCLUDING REMARKS

Stream processing technology enables a large class of applications to consume massive streams and answer queries in near real-time. There however exists general tensions between expressiveness and scalability of the stream processing systems. Our work in this dissertation has made important contributions in increasing both expressiveness and scalability in stream processing, summarized as follows.

- In Cayuga, we developed an expressive query algebra with formal semantics, as well as an efficient and scalable system implementing this algebra.
- In RUMOR, we generalized Cayuga, and unified large-scale stream processing and event processing. RUMOR also provides a platform for integrating future query rewrite based optimization techniques.
- In MMQJP, we developed a large-scale XML stream join processing scheme.

As streaming products with increasingly richer functionality and better performance are continuing to emerge in the market, stream processing will continue to be an important research topic with high practical relevance and impact. Ideas and techniques such as those developed in this dissertation may serve as stepping stone for future research, and may be of commercial importance. It is my hopes that the results in this dissertation, together with many future efforts, will eventually lead to the realization of expressive and large-scale stream processing systems with affordable cost and high reliability.

BIBLIOGRAPHY

[1] Traderbot financial search engine. http://www.traderbot.com/.

[2] Xpath leashed. http://www-db-out.bell-labs.com/user/benedikt/papers/leashed.ps.gz.

[3] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik. The design of the borealis stream processing engine. In *Proc. CIDR*, pages 277–289, 2005.

[4] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. 1995.

[5] R. Adaikkalavan and S. Chakravarthy. Snoopib: Interval-based event specification and detection for active databases. In *Proc. ADBIS*, pages 190–204, 2003.

[6] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Proc. PODC*, pages 53–61, 1999.

[7] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proc. VLDB*, pages 53–64, 2000.

[8] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, 1987.

[9] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software Practice and Experience*, 19(2):171–183, 1989.

[10] A. Arasu, B. Babcock, S. Babu, J. McAlister, and J. Widom. Characterizing memory requirements for queries over continuous data streams. In *Proc. PODS*, pages 221–232, 2002.

[11] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. Technical report, Stanford University, 2003.

[12] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In *Proc. VLDB*, pages 336–347, 2004.

[13] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proc. SIGMOD*, pages 261–272, 2000.

[14] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. PODS*, pages 1–16, 2002.

[15] B. Babcock, M. Datar, and R. Motwani. Load shedding techniques for data stream systems (short paper). In *Proc. Workshop on Management and Processing of Data Streams (MPDS)*, 2003.

[16] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *Proc. ICDE*, pages 350–361, 2004.

[17] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the borealis distributed stream processing system. In *Proc. SIGMOD*, 2005.

[18] Roger Barga, Jonathan Goldstein, Mohamed Ali, and Mingsheng Hong. Consistent streaming through time: A vision for event stream processing. *Proc. CIDR*, 2007.

[19] C. Barton, P. Charles, M. Fontoura, V. Josifovski, D. Goyal, and M. Raghavachari. Streaming xpath processing with forward and backward axes. In *Proc. ICDE*, 2003.

[20] Hans Boehm. Mark-sweep vs. copying collection and asymptotic complexity. ftp://parcftp.xerox.com/pub/gc/complexity.html.

[21] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams — a new class of data management applications. In *Proc. VLDB*, 2002.

[22] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *Proc. VLDB*, pages 606–617, 1994.

[23] C. Y. Chan, P. Felber, M. N. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. In *Proc. ICDE*, pages 235–244, 2002.

[24] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. CIDR*, 2003.

[25] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proc. SIGMOD*, pages 379–390, 2000.

[26] Y. Chen, S. Davidson, and Y. Zheng. An efficient xpath query processor for xml streams. In *Proc. ICDE*, 2006.

[27] Byron Choi. What are real dtds like. 2002.

[28] Coral8. Coral8 home page. http://www.Coral8.com.

[29] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Towards expressive publish/subscribe systems. In *Proc. EDBT*, 2006.

[30] A. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. White. Cayuga: A general purpose event monitoring system. In *Proc. CIDR*, 2007.

[31] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. M. Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *ACM TODS*, 28(4):467–516, 2003.

[32] M. T. Edmead and P. Hinsberg. *Windows NT Performance Monitoring, Benchmarking and Tuning*. Pearson Education, 1998.

[33] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe. In *Proc. SIGMOD*, pages 115–126, 2001.

[34] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. Carey, A. Sundararajan, and G. Agrawal. The bea/xqrl streaming xquery processor. In *Proc. VLDB*, 2003.

[35] A. Galton and J. C. Augusto. Two approaches to event definition. In *Proc. DEXA*, pages 547–556, 2002.

[36] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Composite event specification in active databases: Model and implementation. In *Proc. VLDB*, pages 327–338, 1992.

[37] X. Gong, W. Qian, Y. Yan, and A. Zhou. Bloom filter-based xml packets filtering for millions of path queries. In *Proc. ICDE*, 2005.

[38] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[39] Jim Gray and Andreas Reuter. *Transaction Processing - Concepts and Techniques*. Morgan Kaufmann, 1993.

[40] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata. In *Proc. ICDT*, pages 173–189, 2003.

[41] Todd J. Green, Ashish Gupta, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing xml streams with deterministic automata and stream indexes. *ACM TODS*, 29(4):752–788, 2004.

[42] A. K. Gupta and D. Suciu. Stream processing of XPath queries with predicates. In *Proc. SIGMOD*, pages 419–430, 2003.

[43] M. A. Hammad, M. J. Franklin, W. G. Aref, and A. K. Elmagarmid. Scheduling for shared window joins over data streams. In *Proc. VLDB*, pages 297–308, 2003.

[44] Moustafa A. Hammad, Mohamed F. Mokbel, M. H. Ali, Walid G. Aref, Acc C. Catlin, Ahmed K. Elmagarmid, Mohamed Eltabakh, Mohamed G. Elfeky, Thanaa M. Ghanem, Robert Gwadera, Ihab F. Ilyas, Mirette Marzouk, and Xiaopeng Xiong. Nile: A query processing engine for data streams. *Proc. ICDE*, 2004.

[45] E. N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. B. Park, and A. Vernon. Scalable trigger processing. In *Proc. ICDE*, pages 266–275, 1999.

[46] J. Hopcroft. An n log(n) algorithm for minimizing the states in a finite automaton. In Z. Kohavi, editor, *The Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.

[47] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2nd edition, 2000.

[48] Jeong-Hyon Hwang, Magdalena Balazinska, Alexander Rasin, Ugur Cetintemel, Michael Stonebraker, and Stan Zdonik. High-availability algorithms for distributed stream processing. *Proc. ICDE*, 2005.

[49] Jeong-Hyon Hwang, Ying Xing, Ugur Cetintemel, and Stan Zdonik. A cooperative, self-configuring high-availability solution for stream processing. *Proc. ICDE*, 2007.

[50] IBM. Ibm to acquire cognos to accelerate information on demand business initiative. http://www.ibm.com/press/us/en/pressrelease/22572.wss.

[51] Q. Jiang, R. Adaikkalavan, and S. Chakravarthy. Towards an integrated model for event and stream processing. Technical Report CSE-2004-10, University of Texas at Arlington, 2004. http://www.cse.uta.edu/research/publications/.

[52] J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. In *Proc. ICDE*, 2003.

[53] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. Schema-based scheduling of event processors and buffer minimization for queries on structured data streams. In *Proc. VLDB*, pages 228–239, 2004.

[54] S. Krishnamurthy, M. J. Franklin, J. M. Hellerstein, and G. Jacobson. The case for precision sharing. In *Proc. VLDB*, pages 972–986, 2004.

[55] S. Krishnamurthy, C. Wu, and M. Franklin. On-the-fly sharing for streamed aggregation. In *Proc. SIGMOD*, 2006.

[56] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM TODS*, 6(2):213–226, 1981.

[57] Y. Law, H. Wang, and C. Zaniolo. Query languages and data models for database sequences and data streams. In *Proc. VLDB*, pages 492–503, 2004.

[58] H. Leung and H. Jacobsen. Efficient matching for state-persistent publish/subscribe systems. In *CASCON '03: Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research*, pages 182–196. IBM Press, 2003.

[59] G. Li and H. Jacobsen. Composite subscriptions in content-based publish/subscribe systems. In *Proc. ACM/IFIP/USENIX International Middleware Conference*, 2005.

[60] X. Li and G. Agrawal. Efficient evaluation of xquery over stream data. In *Proc. VLDB*, pages 265–276, 2005.

[61] B. Ludascher, P. Mukhopadhayn, and Y. Papakonstantinou. A transducer-based xml query processor. In *Proc. VLDB*, 2002.

[62] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proc. ICDE*, 2002.

[63] S. R. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proc. SIGMOD*, 2002.

[64] Microsoft. Cedr: Complex event detection and response. http://research.microsoft.com/db/cedr/default.aspx.

[65] I. Motakis and C. Zaniolo. Formal semantics for composite temporal events in active database rules. *Journal of Systems Integration*, 7(3-4):291–325, 1997.

[66] I. Motakis and C. Zaniolo. Temporal aggregation in active database rules. In *Proc. SIGMOD*, pages 440–451, 1997.

[67] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. S. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In *Proc. CIDR*, 2003.

[68] Ed Jr. Pegg. Graph minor. http://mathworld.wolfram.com/GraphMinor.html.

[69] F. Peng and S. Chawathe. Xsq: A streaming xpath engine. *ACM TODS*, 30(2):577–623, 2005.

[70] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in starburst. In *Proc. SIGMOD*, pages 39–48, 1992.

[71] R. Ramakrishnan, D. Donjerkovic, A. Ranganathan, K. S. Beyer, and M. Krishnaprasad. SRQL: Sorted relational query language. In *Proc. SSDBM*, pages 84–95, 1998.

[72] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 3 edition, 2003.

[73] R. Sadri, C. Zaniolo, A. M. Zarkesh, and J. Adibi. Optimization of sequence queries in database systems. In *Proc. PODS*, pages 71–81, 2001.

[74] U. Schreier, H. Pirahesh, R. Agrawal, and C. Mohan. Alert: An architecture for transforming a passive DBMS into an active DBMS. In *Proc. VLDB*, pages 469–478, 1991.

[75] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In Philip A. Bernstein, editor, *Proc. SIGMOD*, pages 23–34, 1979.

[76] T. K. Sellis. Multiple-query optimization. *ACM TODS*, 13(1):23–52, 1988.

[77] P. Seshadri, M. Livny, and R. Ramakrishnan. Sequence query processing. In *Proc. SIGMOD*, pages 430–441, 1994.

[78] P. Seshadri, M. Livny, and R. Ramakrishnan. SEQ: A model for sequence databases. In *Proc. ICDE*, pages 232–239, 1995.

[79] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Proc. ICDE*, pages 25–36, 2003.

[80] Mehul A. Shah, Joseph M. Hellerstein, and Eric A. Brewer. Highly-available, fault-tolerant, parallel dataflows. *Proc. SIGMOD*, 2004.

[81] D. Shasha. *Database Tuning: A Principled Approach*. Prentice Hall, 1992.

[82] A. P. Sistla and O. Wolfson. Temporal conditions and integrity constraints in active database systems. In *Proc. SIGMOD*, pages 269–280, 1995.

[83] Frederick Smith and Greg Morrisett. Comparing mostly-copying and mark-sweep conservative collection. In *ISMM '98: Proceedings of the 1st international symposium on Memory management*, pages 68–78, New York, NY, USA, 1998. ACM Press.

[84] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *Proc. PODS*, pages 263–274, 2004.

[85] U. Srivastava and J. Widom. Memory-limited execution of windowed stream joins. In *Proc. VLDB*, pages 324–335, 2004.

[86] StreamBase. Streambase home page. http://www.streambase.com.

[87] W. Tang, L. Liu, and C. Pu. Trigger grouping: A scalable approach to large scale information monitoring. In *NCA*, pages 148–155, 2003.

[88] Nesime Tatbul and Stan Zdonik. Window-aware load shedding for aggregation queries over data streams. *Proc. VLDB*, 2006.

[89] P. Tucker and D. Maier. Exploiting punctuation semantics in data streams. In *Proc. ICDE*, page 279, 2002.

[90] F. Wang, S. Liu, and Y. Bai P. Liu. Bridging physical and virtual worlds: Complex event processing for rfid data streams. In *Proc. EDBT*, 2006.

[91] H. Wang, C. Zaniolo, and C. Luo. ATLAS: A small but complete SQL extension for data mining and data streams. In *Proc. VLDB*, pages 1113–1116, 2003.

[92] W. White, M. Riedewald, J. Gehrke, and A. Demers. What's "next"? Technical Report TR2006-2033, Cornell University, 2006.

[93] J. Widom and S. Ceri, editors. *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann Publishers, 1996.

[94] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, Saint-Malo (France), 1992. Springer-Verlag.

[95] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *Proc. SIGMOD*, 2006.

[96] Ying Xing, Jeong-Hyon Hwang, Ugur Cetintemel, and Stan Zdonik. Providing resiliency to load variations in distributed stream processing. *Proc. VLDB*, 2006.

[97] Ying Xing, Stan Zdonik, and Jeong-Hyon Hwang. Dynamic load distribution in the borealis stream processor. *Proc. ICDE*, 2005.

[98] A. Yalamanchi, J. Srinivasan, and D. Gawlick. Managing expressions as data in relational database systems. In *Proc. CIDR*, 2003.

[99] R. Zhang, N. Koudas, B. C. Ooi, and D. Srivastava. Multiple aggregations over data streams. In *Proc. SIGMOD*, 2005.

[100] D. Zimmer and R. Unland. On the semantics of complex events in active database management systems. In *Proc. ICDE*, pages 392–399, 1999.